Laxity Prediction and Release in Buffer Optimization of Simulink Models

Abstract

In the Simulink models with single-processor multitask implementation, the time delay and transaction buffers would be caused when the RT blocks are added. Buffer optimization for Simulink models is concerned in this paper. Laxity prediction and laxity release algorithms are proposed for the task scheduling problem. These algorithms allow finding a better solution in the priority assignment procedure. Some experimental results show that with our approach the total system buffer cost could be reduced and system performance has been improved.

1 Introduction

Embedded systems are developing very fast in the modern industry. The development of complex embedded systems is subject to tight cost and performance constraints, thus how to reduce the system cost as well as guarantee the schedulability under strict circumstances is an important problem. Previous works have had analysis on the preemptive scheduling in modelized embedded systems [Scaife and Caspi, 2004] and static-priority scheduling algorithms for multitask problems [Tripakis *et al.*, 2005], but existing code generation tools for the modelization of embedded systems enforces the additional buffering units and latencies between functional blocks, among which some of them are not necessary. A laxity prediction and buffer optimization algorithm has been proposed to complete the priority assignment in [Natale and Pappalardo, 2008], but this algorithm does not consider the different frequencies of functional blocks, thus is not sufficient for buffer optimization. The Mixed Integer Linear Programming (MILP) formulation in [Di Natale *et al.*, 2010] can find out the feasible region for the task mapping, but still did not solve the laxity problem for priority assignment. In this paper we will analyse the laxity prediction problem in the priority assignment procedure of multitask implementations, consequently reduce the buffer cost in the system model. Laxity-release method will also be discussed to improve the system schedulability as well as reduce the buffer cost.

2 Modeling and Simulation in Simulink

Simulink can be used for modeling and simulation of control systems, according to the system synchronous reactive model computation. In Simulink, each block receives one or more inputs and transforms the input into an output [Lee and Varaiya, 2003]. The input signal can be a set of discrete points in time (discrete-time signal), or a continuous time interval (continuous-time signal). Every functional block has a fixed sample time and the base-rate time is the least common multiple number of all the sample time in the system.

There are two different code generation options in the RTW/EC code generator: single task and fixed-priority multitask [Benveniste *et al.*, 2003; Scaife and Caspi, 2004]. For the single task implementation, the execution order is computed by the code generation tool based the topology of the tasks and the partial order deriving from the Simulink semantic rules. Every functional block has a cycle time and new computation will arrive at the start of its cycle time for the functional block to compute. All the functional block computations form a task chain which stands for the sequence of the task executions. The single task implementation requires the the longest task chain will be finished in the base rate time, otherwise the assumption of synchronous reactive model semantics is not met.

For the multitask implementation [Caspi *et al.*, 2008] there can be more than one task chain and every functional block is mapped into a task chain. The execution order in each task is decided by the topology of the tasks and the partial order deriving from the Simulink semantic rules [Agrawal *et al.*, 2004], then same as in single-task implementation, but the difference is that the task may be interrupted and preempted by other tasks with higher priorities, and resumes when the higher-priority tasks finish computing. Every task may be preempted by other tasks for one or more times in a cycle time. The task preemption brings a new problem. As show in Fig. 1, the three blocks are mapped three tasks and each task have one functional block. The first instance of task 2 with period 2 can be interrupted and preempted by the second instance of task 1 with rate 1 because task 1 has a higher priority than task 2, which brings the problem of data consistency and time determinism between these two blocks. The situation can be summarised as follows [Natale and Pappalardo, 2008]:

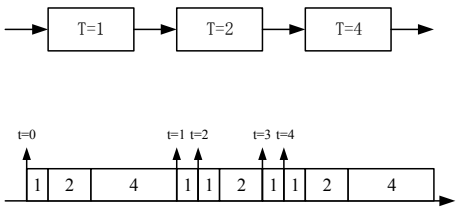

Figure 1: Possible task preemption in multitask models

- Type HL: high-rate (priority) blocks driving low-rate (priority) blocks;
- Type LH: low-rate (priority) blocks driving high-rate (priority) blocks.

Simulink uses Rate Transition (RT) buffers to solve the data consistency and time determinism problem [Baleani *et al.*, 2005; Stuermer *et al.*, 2007]. The RT block acts like a Zero-Holder for the first situation, or a Unit-delay block [Astrom and Wittenmark, 2011] plus a Hold-back for the second situation. We will use type HL RT block and type LH RT block to distinguish the different types of RT buffers.

2.1 Model and Simulation

A simulink model can be represented by a graph $G = \{V, E\}$ with points representing the blocks and edges representing the links between the blocks.

- $V = \{F_1, ..., F_n\}$ is the set of functional blocks. Each block F_i represents a basic computing unit of the system and has one or more inputs and one output. The basic sampling period of F_i is t_i and the Worst-Case-Computing-time is γ_i, which means that signals will come into F_i at the rate of $1/t_i$ for processing. it will take block F_i the time of γ_i to finish the task and gives out the result signal at the same rate.
- $E = \{l_1, ..., l_m\}$ is the set of links. A link $l_i = (F_h, F_k)$ connects the output port of F_h to on of the inputs of F_k. F_h is the source of l_i and F_k the destination, denoted by $F_h = src(l_i)$ and $F_k = dst(l_i)$.
- There may be feedthrough semantics between F_h and F_k and F_h must be finished before F_k, denoted by $F_h \prec F_k$.
- $\tau = \{\tau_1, ..., \tau_l\}$ is the set of tasks. Each task τ_i has an activation period T_i and All the tasks start at the same time $t = 0$. The execution order of task τ_i is decided by its priority π_i and the task with higher priority will have the privilege to execute first.
- $map = (F_i, \tau_j, k)$ is a task mapping relation between the functional block F_i and task τ_j. Each task has several functional blocks assigned to it with the same period time and priority. Functional blocks with different period or different priority to the task will not be able to match into that task. The mapping $map = (F_i, \tau_j, k)$ denotes that F_i is executed in task τ_j with order index of k.
- r_j is the worst-case-response-time of task τ_j, denoting the time of task τ_j that finishes all the functional blocks mapped into it. With c_j denoting the worst-case-computing-time of τ_j, r_j can be computed by the following formula [Joseph and Pandya, 1986]:

$$r_j = c_j + \sum_i \lceil \frac{r_j}{t_i} \rceil c_i \tag{1}$$

- $S(F_i)$ denotes the *synchronous set* of functional block F_i. It represents the transitive closure of the immediate predecessor and successor relations of blocks with the same rate. The set $S(F_i)$ can be constructed as follows: firstly $S(F_i) = F_i$, then at each step for each F_j in $S(F_i)$, if the immediate predecessors and successors have the same rate with F_j, add it to $S(F_i)$. The procedure ends when no more functional blocks can be added to the set. All the functional blocks in one synchronous set share the same rate and are linked together by edges between the blocks in the set. Thus the entire graph can be denoted as $S = (S_1, S_2, ..., S_m)$. S_i has a period time of T_i and $C_i = \sum_{F_l \in S_i} c_l$ is the sum of worst-case computation time of all the functional blocks in S_i.

2.2 Laxity Upper and Lower Bound

As [Natale and Pappalardo, 2008] pointed out, The sum of the computation time of all the members in each set $path(S_i)$ is a lower bound for the worst-case computation time of S_i. The upper bound laxity of S_i is defined as follows:

$$l_{upper_o,i} = min_{S_k}(T_k - max_{path(S_k)} \sum_{S_l} C_l) \tag{2}$$

where $S_l \in path(S_k)$ and $S_k \in \{succ(S_i) \cup S_i\}$.

The lower bound is not sufficient for calculating laxity. Suppose there are only two function blocks with different rate. Function block 1 has a period time of 1 and its worse-case computing time is c_1. Function block 2 is block 1's successor, which has a period time of 2 and worse-case computing time c_2. The premises here are that $c_1 \leq 1$ and $c_2 \leq 2$. Since block 2 is block 1's successor and has a bigger period time than block 1, its processing may be preempted by block 1. There are mainly three execution results for different c_1 and c_2.

- Case 1 ($c_2 \leq c_1$): Start from $t = 0$, block 1 starts to execute and at $t = c_1$ block 1 finishes computing, and then block 2 starts its execution. At $t = c_1 + c_2 \leq 1$, block 2 finishes its execution and a new process of block 1 has not come. The processer is idle until $t = 1$ when a new process request of block 1 comes. At $t = c_1 + 1$, block 1 finishes its execution, and both block 1 and block 2 finishes all their execution.
- Case 2 ($c_1 + c_2 > 1$ & $2c_1 + c_2 \leq 2$): Start from $t = 0$, block 1 starts to execute and at $t = c_1$ block 1 finishes computing, and then block 2 starts its execution. At $t = 1$, block 2 does not finish its execution as a result of $c_1 + c_2 > 1$, but a new process of block 1 comes, since it has a period time of 1. The execution of block 2 will be preempted by block 1. At $t = c_1 + 1$, block 1 finishes

execution and block 2 continues to execute till $t = 2c_1 + c_2 <= 1$.

- Case 3 ($1 < c_1+c_2 \leq 2 \& 2c_1+c_2 > 2$): As $c_1+c_2 > 1$, block 2 will not finish execution at $t = 1$, thus will be preempted by a new block 1 execution. After the second execution of block 1, block 2 will continue to execute and finish at $t = 2c_1 + c_2$. As a result of $2c_1 + c_2 > 2$, block 2 did not meet its deadline and the system is not feasible.

If we use Equation. 2 to calculate the laxity of the two blocks, then $l_1 = c_1$ and $l_2 = c_1+c_2$. Let's consider the three cases above. Block 1 has the highest priority and will preempt any other blocks when a new block 1 execution comes. So any execution of block 1 starting at t_s will finish at $t_s + c_1$. In case 1, the execution of block 2 may start at $t = 0$ but is preempted by block 1, so it will start at $t = c_1$ and finishes at $t = t_1 + c_2$. The result is the same as Equation. 2 gives. But in case 2 and case 3 we have different results. Both case 2 and case 3 gives a result of $l_2 = 2c_1 + t2$. In case 2, $2c_1 + c_2 \leq 2$, which means block 2 meets its deadline. In case 3 the execution of block 2 finishes at $t = 2c_1 + c_2 > 2$ and exceeds its deadline, but the Equation. 2 gives a laxity prediction of $l_{upper,2} = t_2 - (c_1 + c_2) > 0$, thus cannot present the actual situation.

We call it a frequency-doubling *FD* situation when for any two blocks S_i and S_j with block period time T_i and T_j in the system, if $T_i < T_j$ then T_i must be integral division of T_j. The above case shows how the Equation. 2 fails to calculate the correct laxity in the FD situation. The situation is more complicated when the period of one block is not the integral multiple of the other blocks, which we call non-frequency-doubling *NFD* situation.

We define the lower bound of the laxity as

$$l_{lower,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k}{T_l} \rceil C_l}{T_k}) \quad (3)$$

$pre(S_k) = \{S_j | \pi_j \geq \pi_k : \pi_j \; is \; priority \; of \; S_j\}$ contains all the block sets with higher priority than S_k, $Pre(S_k) = \{pre(S_k) \cup S_k\}$, $S_k \in \{succ(S_i) \cup S_i\}$.

$\lceil \frac{T_k}{T_l} \rceil$ implies that

- $T_k > T_l$: block S_l will be computed $\frac{T_k}{T_l}$ times in one cycle time of S_k as a result of that T_l is a integral diversion of T_k;
- $T_k = T_l$: block S_l will be computed $\frac{T_k}{T_l} = 1$ time in one cycle of S_k;
- $T_k < T_l$: Although S_k has a shorter period sample time than S_l, block S_l have to finish computing for one time in the cycle time of S_k, where $\lceil \frac{T_k}{T_l} \rceil = 1$ in Equation. 3

Here we did not use $path(S_k)$ because the laxity of S_i does not only depend on the block sets in $path(S_i)$ but also other block sets in other paths. All the block sets with high priority than S_k will contribute to the laxity of S_i and should be counted in when computing laxity. When we have to calculate the laxity of S_i, we suppose that there are n synchronous sets in total, among which n_i numbers of sets' priorities have been decided, from priority n as the highest to priority $n - n_i + 1$ as the lowest. Then for S_i, R_k should be that we give priority $n - n_i$ to S_i and calculate its response time in all the $n_i + 1$ sets. If S_j is the successors of S_i, then we will give priority $n - n_i$ to S_i and $n - n_i - 1$ to S_j to compute the response time of S_j.

As for different S_k, T_k will be different, which may bring system deviation to the laxity result, so the laxity is normalized by being divided by T_k.

In the same way, the upper bound of laxity is defined as

$$l_{upper,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}) \quad (4)$$

2.3 Use Response Time for Laxity Prediction

Suppose block 1 has a period time of 2 and execution time c_1 and its successor block 2 has a period time of 3 and execution time c_2. The base rate is 1 and the lowest common multiple number of their period is 6. If $c_1 = 1.2$ and $c_2 = 0.7$, from $t = 0$ to $t = 6$, block 1 has executed 3 times, each take c_1 time to compute. Block 2 has executed 2 times. In the first execution of block 2, it is preempted by block 1 for one time and finish at $t = 1.9$ after it starts. In the second execution block 2 is preempted one time and finished at $t = 3.9$ and the laxity should be $l_2 = min\{3 - 1.9, 3 - 0.9\} = 1.1$. But the laxity given by the lower bound is $l_{lower,2} = 3 - 0.7 - 1.2 * 2 = -0.1 < 0$, meaning that block 2 didn't finish all the computing before its deadline, while in fact it did. The upper bound of laxity failed to represent the accurate laxity because block 2 finished its computing before the second occurrence of block 1, while the upper bound of laxity still counted in the computing time of the second occurrence of block 1.

The accurate laxity is between the lower bound and upper bound, and can be represented by the response time of the tasks [Sjodin and Hansson, 1998].

$$l_{resp,i} = min_{S_k}(1 - \frac{R_k}{T_k}) \quad (5)$$

where $S_k \in \{succ(S_i) \cup S_i\}$ and R_k denotes the response time of S_k in the current priority assignment system.

Algorithm 1 Laxity Prediction Algorithm

Input:
Set S_i for laxity calculation
Output:
l_i as the laxity of S_i and the corresponding tail set S_k

```
l_i ← ∞
for all S_l in {succ(S_i) ∪ S_i} do
    R_l ← CalcResponseTime(S_l)
    laxity ← 1 − R_l/T_l
    if laxity < l_i then
        l_i ← laxity, S_k ← S_l
    end if
end for
```

3 The optimization procedure

The optimal solution mainly depends on the mapping of the synchronous sets into tasks and the priority assignment, and

other arrangements like the order with which the blocks are scheduled inside each task and the order of tasks in each synchronous sets may also count. It is too difficult or even practically impossible to consider all these degrees of freedom at once in a search procedure. [Natale and Pappalardo, 2008] proposes a two-stage optimization search procedure to reach the minimum type LH buffers.

The task mapping procedure will use the rate-monotonic scheduling algorithm to map sets into blocks and decide the priority for each task. Every time RM procedure begins, the algorithm will get all the sets with no incoming links. The R-M scheduling algorithm requires that the set with the highest sampling rate should have the highest priority. If there is only one set with the highest sampling rate, it will get the highest rate; but if there is more than one set with the same highest sampling rate, RM scheduling algorithm can not decide the priority. Laxity prediction method is used to decide which set should have the highest rate. As Equation. 5 indicates that laxity refers to the time remaining after completing the task chain, the smaller the laxity time is, the harder that task can be scheduled. So the task set with the least laxity must have the highest priority. By applying the RM algorithm and the laxity prediction algorithm, the priorities of the task sets can be determined and the schedulability of the task mapping can be checked.

The first stage is to applying the RTW translation rules to evaluate the schedulability of the system. By adding an RT block and possibly a delay on every link upon which a dependency constraint among two blocks with different rates exist. All the synchronous sets with the same rate are mapped to the same task and the standard rate-monotonic fixed-priority analysis is performed on the task sets. Consider the optimality of the rate-monotonic priority assignment, if this solution is not schedulable, there is no other fixed-priority assignment which can make the set schedulable, and the scheduling problem on that graph has no solution. If the solution is schedulable, its buffer cost becomes an upper bound for the cost of the optimal solution. This solution is called RTW solution.

If the RTW solution is schedulable, we will test the schedulability of the graph with no type LH transaction buffers, on the other extreme. This solution is called No Type LH mapping solution. If the task mapping under this solution is schedulable, then there is no room for optimization because there is no type LH buffers to remove and the minimum buffer cost should be the cost of buffers in No Type LH mapping solution.

If the RTW solution is schedulable but the No Type LH mapping solution is not, the search stage begins. A branch-and-bound procedure starts from the RTW solution and the No Type LH mapping solution to search for an optimization mapping solution. The breath-first search procedure is describe in [Natale and Pappalardo, 2008]

4 Laxity Release Optimization

As we can see from above, the laxity of one functional block in some way indicates the difficulty of the schedulability of that task associated with that functional block in the system. The lower the laxity be, the more the task can finish before its deadline. If one block has a laxity less than zero, that task will not be able to finish before its deadline, thus the system is not schedulable. We call a task τ_i to be a *Dangerous Task* if the laxity of that functional block meets the condition $1 - l_i > laxityGuard$, $laxityGuard \in \mathbb{R}$ and $laxityGuard \in (0, 1)$, in which $laxityGuard$ is a user defined real number to act as a threshold value. If the laxity of one block is in the dangerous zone, we can use *Laxity Release* method to increase the laxity, thus improve the schedulability of the tasks. If the worst-case-computing time is fixed as a experimental parameter for tasks, only the frequency time can be adjusted.

Algorithm 2 Tail Release Algorithm for Laxity Release

Input:
Set S_i for release, laxity safe guard value $laxityGuard$
Output:
l_i as the laxity of S_i after laxity release
1: $laxity_i, S_k \leftarrow LaxityPrediction(S_i)$
2: **while** $1 - laxity_i > laxityGuard$ **do**
3: $T_k \leftarrow 2T_k$;
4: $UpdateGraph(S_k)$;
5: $laxity_i, S_k \leftarrow LaxityPrediction(S_i)$
6: **end while**

4.1 Laxity Release in Upper and Lower Bound Calculation

For the upper bound of laxity $l_{upper,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k})$, changing T_k will directly changing the laxity result. For the lower bound of laxity $l_{lower,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k}{T_l} \rceil C_l}{T_k})$, considering the ceiling function $\lceil \frac{T_k}{T_l} \rceil$, there are two kinds of laxity release methods.

Tail Release

For all the $S_l \in pre(S_k)$, if $T_k < T_l$, $l_{FD,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k})$, the laxity has no relation with T_l. Increasing T_l for $S_l \in pre(S_k)$ will not change the laxity, so the only way of release the laxity is to Increasing T_k.

- Case 1: If T_k is doubled to $T_k^{'} = 2T_k$ and still meets the condition $T_k^{'} < T_l$ for every $S_l \in pre(S_k)$, then $\frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k^{'}} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, laxity of S_i is enlarged, or *released*.
  More generally, $T_k^{'} = mT_k$, $m \in \{2^{\mathbb{N}}\}$ and still $T_k^{'} < T_l$ for every $S_l \in pre(S_k)$, $\frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k^{'}} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, laxity of S_i is enlarged, or *Released*.
  Noticing that here we make $m \in \{2^{\mathbb{N}}\}$ so that the system still meets the FD situation requirement after laxity release process.
- Case 2: if $T_k^{'} = mT_k \geq T_l$ for every $S_l \in pre(S_k)$, $m \in \{2^{\mathbb{N}}\}$. Noticing that $T_k < T_l$ for all the

$S_l \in pre(S_k)$, in *FD* situation, the only possibility is that $T_l = m^{'}T_k, m^{'} \in \{2^{\mathbb{N}}\}, m^{'} < m$ for every $T_l \in pre(S_k)$. Thus $\lceil \frac{T_k^{'}}{T_l} \rceil = \lceil \frac{mT_k}{m^{'}T_k} \rceil = \frac{m}{m^{'}} > 1$ for $S_l \in pre(S_k)$, $\frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k^{'}}{T_l} \rceil C_l}{T_k^{'}} = \frac{\sum_{S_l \in pre(S_k)} \frac{m}{m^{'}} C_l + C_k}{mT_k} = \frac{\sum_{S_l \in pre(S_k)} \frac{1}{m^{'}} C_l + \frac{C_k}{m}}{T_k} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, laxity of S_i is released.

- Case 3: If for some but not all $S_l \in pre(S_k)$, $T_k^{'} = mT_k \geq T_l$, $m \in \{2^{\mathbb{N}}\}$. Let $T_k = T_l - \delta_k$, $\delta_k \geq 0$, $\lceil \frac{T_k^{'}}{T_l} \rceil = \lceil \frac{mT_l - m\delta_k}{T_l} \rceil \leq m$, $\frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k^{'}}{T_l} \rceil C_l}{T_k^{'}} \leq \frac{\sum_{S_l \in A} mC_l + \sum_{S_l \in B} C_l}{mT_k} = \frac{\sum_{S_l \in A} mC_l + \sum_{S_l \in B} \frac{C_l}{m}}{mT_k} = \frac{\sum_{S_l \in Pre(S_k)} C_l - \sum_{S_l \in B} \frac{(m-1)C_l}{m}}{T_k} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, laxity of S_i is released.

In the above three cases, all the laxity of S_i can be release by enlarging the period time of S_k from T_k to mT_k, $m \in \{2^{\mathbb{N}}\}$. As S_k has the smallest priority in $Pre(S_k)$, this laxity release method is called *Tail Release*. Tails release algorithm is shown in Algorithm. 2.

Algorithm 3 Internal Release Algorithm for Laxity Release

Input:
Set S_i for release, laxity safe guard value $laxityGuard$
Output:
l_i as the laxity of S_i after laxity release

```
laxity_i, S_k ← LaxityPrediction(S_i)
Pre_{S_k} ← {S_j | π_j ≥ S_k}
while 1 − laxity_i > laxityGuard do
    S_p ← FindLeastPeriodTimeTask(Pre_{S_k})
    T_p ← 2T_p;
    UpdateGraph(S_k);
    laxity_i, S_k ← LaxityPrediction(S_i)
end while
```

Internal Release

If $T_k \geq T_l$ for all the $S_l \in Pre(S_k)$, then $l_{FD,i} = min_{S_k}(1 - \sum_{S_l \in Pre(S_k)} \frac{C_l}{T_l})$. We can see that laxity will be released if we use the *Tail Release* method to increase T_k. Except for *Tail Release* there is another way to adjust the laxity. For any $S_p \in pre(S_k)$, $T_p \leq T_k$, increase T_p to $T_p^{'} = mT_p$ and analyse how the laxity changes.

- Case 1: $T_p^{'} = mT_p \leq T_k$, $m \in \{2^{\mathbb{N}}\}$, then $l_{FD,i}^{'} = min_{S_k}(1 - \sum_{S_l \in Pre(S_k)} \frac{C_l}{T_l} + \frac{C_p}{T_p} - \frac{C_p}{mT_p}) > l_{FD,i}$ and the laxity is released.
- Case 2: $T_p^{'} = mT_p > T_k$, $m \in \{2^{\mathbb{N}}\}$, then $l_{FD,i}^{'} = min_{S_k} 1 - \frac{\sum_{S_l \in Pre(S_k)}}{T_k} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} \frac{T_k}{T_l} C_l - \frac{T_k}{T_p} C_p + C_p}{T_k}) =$ $min_{S_k}(1 - \sum_{S_l \in Pre(S_k)} \frac{C_l}{T_k} + \frac{T_k - T_p}{T_p T_k} C_p) > l_{FD,i}$. The laxity is released.

Algorithm 4 Find the Least Period Time Set

Input:
Set $Pre\{S_k\}$ for search
Output:
S_p as the least period time set in Pre_{S_k}

```
min_T = ∞, collection SP = ∅
for all S_i in Pre_{S_k} do
    SP ← FindLeastPeriodTimeSets(Pre_{S_k})
end for
if Count(SP) = 1 then
    return RandomPick(SP)
else
    collection SC = ∅
    SC ← FindMaxComputingTimeSets(SP)
    S_p ← RandomPick(SC)
    return S_p
end if
```

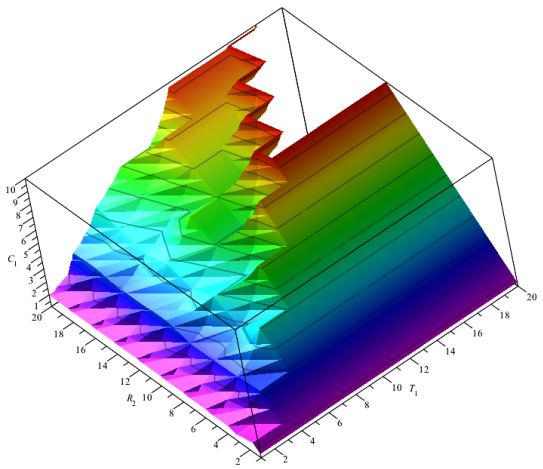

Figure 2: R_2-T_1-C_1 relation graph

This method is called *Internal Release*. It is performed by finding the least period time task set S_p and increasing t_p to release the laxity. If there are more than one task sets with the same least period time, select the task set S_p with the biggest worst-case-computing time and increase t_p to release the laxity. Internal release algorithm is show in Algorithm. 3.

If there $\exists S_l \in Pre(S_k)$ so that $T_k \geq T_l$, and $\exists S_l^{'} \in Pre(S_k)$ so that $T_k < T_l^{'}$, both the tail release and the internal release methods can be used to release the laxity.

4.2 Laxity Release in Response Time Calculation

In the response time calculation of laxity $l_{resp,i} = min_{S_k}(1 - \frac{R_k}{T_k})$, $S_k \in succ\{\{S_i\} \cup S_i\}$. If $R_k \leq T_k$, task set S_k can finish computing before its deadline. For $R_k = \sum_{S_l \in pre(S_k)} \lceil \frac{R_k}{T_l} \rceil C_l + C_k$, tail release method can always be applied for laxity release purpose. We will show how internal release can be applied to $l_{resp,i}$.

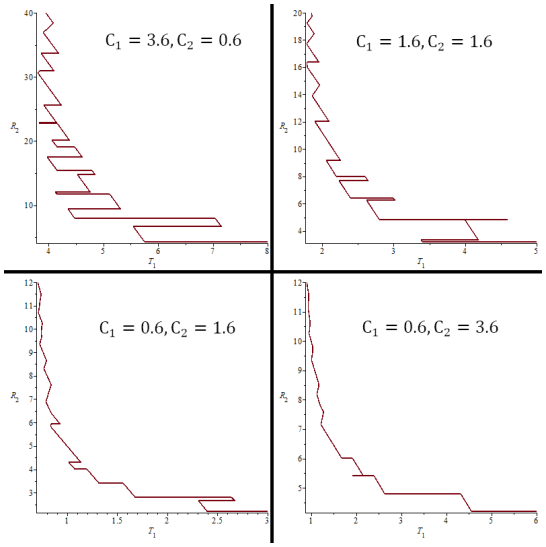

Figure 3: R_2-T_1 relation graph

For block S_1 with T_1, C_1 and its successor S_2 with T_2, C_2, $R_2 = \lceil \frac{R_2}{T_1} \rceil C_1 + C_2$. C_2 is a constant number and the numerical value relationship between R_2, T_2, C_2 is show as Fig. 2. Response time is defined as the minimum nonnegative number that fulfills the equation. As shown in Fig. 3, for different C_1 and C_2, R_2 is tending to decrease as T_1 increases. This result can be extended to general condition $R_k = \sum_{S_i \in \{pre(S_k) - S_l\}} \lceil \frac{R_k}{T_i} \rceil + \lceil \frac{R_k}{T_l} \rceil C_l + C_k$. Generally, the increase of T_l will lead to the decrease of S_k, thus internal release can be applied to response time calculation of laxity.

4.3 Laxity Release Procedure

So we propose the *Laxity Release Procedure* as follows: in the priority assignment procedure, compute the laxity of every candidate tasks. If task S_k has a l_k that $1 - l_k > laxityGuard$, search in the tasks with priority greater than S_k with the minimum frequency time and enlarge the frequency time of that task. Repeat this procedure until every candidate task S_k fulfills $1 - l_k \leq laxityGuard$ and assign priority using RMS algorithm.

The parameter $laxityGuard$ is a threshold value and can be adjusted to meet different practical needs. As laxity and system utilization factor both characterized the schedulability of the system in different point of view, the threshold $laxityGuard$ can be set to the system unitization.

5 Experimental Results

Table 1: Experimental Results for different utilizations

Utilization	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85
$l_{\text{upper}_\text{o}}$ buf	0	0	4	14	14	28	36	64
l_{upper} buf	0	0	4	10	12	24	28	62
l_{lower} buf	0	0	4	10	12	24	28	56
l_{resp} buf	0	0	4	10	12	20	26	46
release buf	0	0	4	10	12	20	26	40

Table 2: Experimental Results for utilizations=0.85

Test Cases												sum
$l_{\text{upper}_\text{o}}$ buf	0	4	12	8	6	2	2	8	4	6	6	64
l_{upper} buf	0	4	12	8	6	2	2	8	4	6	6	4	62
l_{lower} buf	0	4	6	8	6	2	2	8	4	6	6	4	56
l_{resp} buf	0	4	4	6	2	2	2	8	4	6	4	4	46
release buf	0	2	4	4	2	2	2	8	4	6	2	4	40

To evaluate the algorithm performance we will compare the buffer cost by using different laxity prediction algorithms and laxity release algorithm. The experimental results have been performed by generating random graphs with characterized rules. As the RT buffer only exists between functional blocks with different rates, we only generate random synchronous set graphs, which means every two vertexes directly connected by one edge have two different rates. Each graph 2 source blocks with 15 synchronous sets each. The utilization factor ranges from 0.5 to 0.85. The possible sampling rates of the graphs are the base rate $\times 2$, the base rate $\times 3$ and the base rate $\times 5$; Each graph set contains 12 randomly generated graphs and buffer optimization procedure is performed on the graphs. The type LH RT-transaction buffer cost is fixed at 2.

Table 1 shows the result of adding all the buffer cost in each graph set for different utilizations. Original laxity upper bound $l_{upper_o,i}$ from 2, normalized upper bound $l_{upper,i}$ and lower bound $l_{lower,i}$, response time calculation $l_{resp,i}$ algorithms as well as laxity release algorithm are used. We can see that for low CPU utilizations, all the five methods give out a very low buffer cost. As the utilization grows, buffer cost grows obviously, and the latter four methods can generate a better result, i.e., lesser buffer cost, than $l_{upper_o,i}$. When the utilization reaches 0.85 and system laxity is relatively low, laxity release methods coming into effect. Table 2 shows the detailed result of the 12 test cases in graph set with $utilization = 0.85$. With $l_{resp,i}$ algorithm generates a better result than $l_{upper_o,i}$, $l_{upper,i}$ and $l_{lower,i}$, laxity release method can give out an even lesser buffer cost than $l_{resp,i}$.

6 Conclusion

This paper analysed the laxity prediction and release problem in the buffer optimization procedure of multitask simulink models. The research shows the possibility to improving the performance of the laxity prediction thus to reduce the system cost of multitask simulink implementations. Experiments show with our approach of laxity prediction based on response time the performance of priority assignment is improved. Also we discussed the laxity release problem in the priority assignment procedure for high utilization systems. Experiments show how the laxity prediction algorithm can find the dangerous task and release the laxity to improve the system schedulability as well as reduce the system buffer cost.

References

[Agrawal *et al.*, 2004] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, 2004.

[Astrom and Wittenmark, 2011] K.A. Astrom and B. Wittenmark. *Computer-controlled systems: theory and design*. Dover Publications, 2011.

[Baleani *et al.*, 2005] M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni-Vincentelli. Efficient embedded software design with synchronous models. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 187–190. ACM, 2005.

[Benveniste *et al.*, 2003] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[Caspi *et al.*, 2008] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):15, 2008.

[Di Natale *et al.*, 2010] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of multitask implementations of simulink models with minimum delays. *Industrial Informatics, IEEE Transactions on*, 6(4):637–651, 2010.

[Joseph and Pandya, 1986] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[Lee and Varaiya, 2003] E.A. Lee and P. Varaiya. *Structure and interpretation of signals and systems*. Lee & Seshia, 2003.

[Natale and Pappalardo, 2008] M.D. Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):23, 2008.

[Scaife and Caspi, 2004] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time-and event-triggered systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 119–126. IEEE, 2004.

[Sjodin and Hansson, 1998] M. Sjodin and H. Hansson. Improved response-time analysis calculations. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 399–408. IEEE, 1998.

[Stuermer *et al.*, 2007] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper. Systematic testing of model-based code generators. *Software Engineering, IEEE Transactions on*, 33(9):622–634, 2007.

[Tripakis *et al.*, 2005] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 353–360. ACM, 2005.