Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Laxity Release Optimization for Simulink Models

Kaijie Qin, Guiming Luo, Xibin Zhao

School of Software, Tsinghua University, Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China (Email: danielqkj@gmail.com, gluo@mail.tsinghua.edu.cn).

Abstract: In Simulink models with single-processor multitask implementation, time delays and transaction buffers emerge when Rate Transition (RT) blocks are added. This paper examines Simulink modeling and buffer optimization. The concept of laxity release is defined for the priority assignment procedure. The algorithms of laxity prediction and laxity release are proposed for the task scheduling problem. The laxity bounds and laxity release bounds are obtained, and the response time based on laxity release is calculated. Some experimental results are given to show that with our approach, total system buffer costs are reduced and system performance is improved.

Keywords: Discrete event modeling and simulation; Task scheduling of hybrid systems; Model predictive control of hybrid systems

1. INTRODUCTION

Embedded systems are developing very quickly in modern industry. System resource cost and performance issues are very import to the development of complex embedded systems. Thus, reducing system cost and guaranteeing schedulability under strict circumstances are important issues to address. Previous work has analyzed preemptive scheduling in modelized embedded systems (Scaife and Caspi, 2004) and static-priority scheduling algorithms for multitasking problems (Tripakis et al., 2005). A laxity prediction and buffer optimization algorithm used to complete the priority assignment was proposed in (Natale and Pappalardo, 2008), but this algorithm did not consider the different frequencies of functional blocks. Although Mixed Integer Linear Programming (MILP) in (Di Natale et al., 2010) can determine the feasible region for task mapping, it still cannot solve the laxity problem for priority assignment. In this paper, we analyze the laxity prediction problem in the priority assignment procedure of multitask implementation. The laxity release method is discussed in terms of improving system schedulability, as well as reducing the buffer cost.

1.1 Modeling and Simulation in Simulink

Simulink can be used for the modeling and simulation of control systems, and it is based on the synchronous reactive model of computation. In Simulink, every functional block has a fixed sample time, and the base-rate time is the least common multiple of all sample times in the system.

There are two different code generation options in the RTW/EC code generator: single-task and fixed-priority multitasking (Benveniste et al., 2003; Scaife and Caspi, 2004). For single-task implementation, the execution order is computed by the code generation tool based on the topology of the tasks and the partial order derived from the Simulink semantic rules. All the functional block computations form a task chain that represents the sequence of task executions. Single-task implementation requires that the longest task chain be finished in the base-rate time; otherwise, the assumption of synchronous reactive model semantics is not met.

For 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 derived from the Simulink semantic rules (Agrawal et al., 2004). The same is true in single-task implementation, but the difference here is that the task may be interrupted and preempted by other tasks with higher priorities, and will resume when the higher-priority tasks finish computing. Every task may be preempted by other tasks one or more times in one cycle time. Task preemption results in a new problem. The situation can be summarized as follows (Natale and Pappalardo, 2008):

- 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 determination problems (Baleani et al., 2005; Stuermer et al., 2007). The RT block acts like a Zero-Order Hold block 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 blocks and type LH RT blocks to distinguish among the different types of RT buffers.

⋆ This work was supported by the Funds NSFC61171121, 863 Project 2012AA040906, and the Science Foundation of Chinese Ministry of Education - China Mobile 2012.

This paper is structured as follows: Section I provides the necessary background related to the schedulability and buffer optimization problems in Simulink models. Section II will outline the implementation, including the formulation, of Simulink models. The lower bound, upper bound, and response time calculation of the laxity are presented in this section. Next, in Section III, the laxity release algorithms are introduced and demonstrated. In Section IV, we benchmark our algorithm versus existing methods, and Section V concludes the paper.

2. SIMULINK MODELING

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; it 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 generate the result signal.
- $E = \{l_1, ..., l_m\}$ is the set of links. A link $l_i = (F_h, F_k)$ connects the output port of F_h to one of the inputs of F_k. F_h is the source of l_i and F_k is the destination, denoted by $F_h = src(l_i)$ and $F_k = dst(l_i)$, respectively.
- There may be feed through semantics between F_h and F_k, and F_h must be finished before F_k, which is 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; the tasks with higher priority will have the privilege to execute first.
- $map = (F_i, \tau_j, k)$ is a task mapping relation between 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 periods or priorities from 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 an 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):$r_j = c_j + \sum_i \lceil \frac{r_j}{t_i} \rceil c_i$, where the index i spans over higher-priority tasks τ_i ($\pi_i \geq \pi_j$).
- $S(F_i)$ 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:

  First, $S(F_i) = F_i$. Then, at each step for each F_j in $S(F_i)$, if the immediate predecessors and successors have the same rate as 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 the worst-case computation times of all the functional blocks in S_i.

For each set S_i, $succ(S_i)$ defines the set of all its successor sets. If $S_j \in succ(S_i)$, there $\exists F_h \in S_i$, $F_l \in S_j$, and $l_i = (F_h, F_l) \in E$. Furthermore, $path(S_i)$ is defined as $path(S_i) = S_{i_0}, S_{i_1}, ..., S_{i_k}$ a set of synchronous sets with the following properties: S_{i_0} has no incoming link, $S_{i_k} = S_i$, and there exists (at least) a link s_l connecting each pair S_{i_j}, $S_{i_{j+1}}$, where $j = 0..k-1$. For each set S_i, there can be multiple $path(S_i)$ sets.

Laxity is a key concept in this paper. The idea of proposing laxity is to give an evaluation of the schedulability of tasks. The schedulability of one task has a close relationship with the time remaining in the period time after the task finishes computing.

As (Natale and Pappalardo, 2008) pointed out, the sum of the computation times 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:

$$l_{upper_o,i} = min_{S_k}(T_k - max_{path(S_k)} \sum_{S_l} C_l) \qquad (1)$$

where $S_l \in path(S_k)$ and $S_k \in \{succ(S_i) \cup S_i\}$. The o in the $l_{upper_o,i}$ means that the laxity in Equation 1 is the original formula from previous research. In this paper, other laxity calculation methods will be proposed and compared with the original.

From Equation 1, we can see that when calculating the laxity of S_i, the time remaining after execution for both S_i and all the tasks in $succ(S_i)$ will be considered.

The lower bound is not sufficient for calculating laxity. Suppose there are only two function blocks with different rates. Function block F_1 has a period time of 1, and its worst-case computing time is c_1. Function block F_2 is F_1's successor, which has a period time of 2 and a worst-case computing time c_2. The premises here are that $c_1 \leq 1$ and $c_2 \leq 2$. Since F_2 is F_1's successor and has a greater period time than F_1, its processing may be preempted. There are three main execution results for different c_1 and c_2.

- Case 1 ($c_2 \leq c_1$): Start from $t = 0$; F_1 starts to execute, and at $t = c_1$, F_1 finishes computing. Then, F_2 starts its execution. At $t = c_1 + c_2 \leq 1$, F_2 finishes its execution, but a new process request of F_1 has not yet arrived. The processer is idle until $t = 1$, when a new process request of F_1 arrives. At $t = c_1 + 1$, F_1 finishes its execution, and both F_1 and F_2 have therefore finished all their executions.
- Case 2 ($c_1 + c_2 > 1$ & $2c_1 + c_2 \leq 2$): Start from $t = 0$; F_1 starts to execute, and at $t = c_1$, F_1 finishes computing. Then, F_2 starts its execution. At $t = 1$, F_2 does not finish its execution as a result of $c_1 + c_2 > 1$, but a new task for F_1 arrives, since it has a period time of 1. The execution of F_2 will be preempted. At $t = c_1 + 1$, F_1 finishes its execution, and F_2 continues to execute until $t = 2c_1 + c_2 <= 1$.

- Case 3 ($1 < c_1 + c_2 \leq 2 \& 2c_1 + c_2 > 2$): As $c_1 + c_2 > 1$, F_2 will not finish its execution at $t = 1$. Thus, it will be preempted by a new execution by F_1. After the second execution of F_1, block 2 will continue to execute and will 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 cannot be considered feasible.

If we use Equation 1 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. F_1 has the highest priority and will preempt any other blocks when a new F_1 execution arrives. Therefore, any execution of F_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 F_1, so it will start at $t = c_1$ and finish at $t = t_1 + c_2$. The result is the same as that given by Equation 1. But in case 2 and case 3, we have different results. Both case 2 and case 3 give a result of $l_2 = 2c_1 + t_2$. 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 Equation 1 gives a laxity prediction of $l_{upper_o,2} = t_2 - (c_1 + c_2) > 0$. Thus, it cannot represent the actual situation.

What we call a frequency-doubling (*FD*) situation is defined as follows: for any two blocks in the system S_i and S_j with block period times of T_i and T_j, respectively, if $T_i < T_j$, then T_i must be an integral division of T_j. The above case shows how Equation 1 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 other blocks, which we call a non-frequency-doubling (*NFD*) situation.

We define the lower bound of laxity as

$$l_{lower,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k}{T_l} \rceil C_l}{T_k}) \tag{2}$$

$pre(S_k) = \{S_j | \pi_j \geq \pi_k : \pi_j \ is\ priority\ of\ S_j\}$ 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

- $T_k > T_l$: T_l is a integral division of T_k, and as a result block S_l will be computed $\frac{T_k}{T_l}$ times in one cycle time of S_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 has to finish computing for one time in the cycle time of S_k, where $\lceil \frac{T_k}{T_l} \rceil = 1$ in Equation 2

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 a higher priority than S_k will contribute to the laxity of S_i, and should be considered 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 the 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 mean 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 successor of S_i, then we will give priority $n - n_i$ to S_i and priority $n - n_i - 1$ to S_j to compute the response time of S_j.

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

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

$$l_{upper,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}) \tag{3}$$

Suppose F_1 has a period time of 2 and execution time c_1, and its successor F_2 has a period time of 3 and execution time c_2. The base rate is 1 and the lowest common multiple of their periods is 6. If $c_1 = 1.2$ and $c_2 = 0.7$, from $t = 0$ to $t = 6$, F_1 has executed 3 times, each taking c_1 time to compute, while F_2 has executed 2 times. In the first execution of F_2, it is preempted by F_1 one time and finishes at $t = 1.9$ after it starts. In the second execution, F_2 is preempted one time and finishes at $t = 3.9$, and the laxity should be $l_2 = min\{1 - 1.9/3, 1 - 0.9/3\} = 0.3667$. But the laxity given by the lower bound is $l_{lower,2} = 1 - (0.7 - 1.2 * 2)/3 = -0.0333 < 0$, meaning that F_2 did not finish all the computing before its deadline, while in fact it did. The upper bound of laxity failed to represent the accurate laxity because F_2 finished its computing before the second occurrence of F_1, while the upper bound of laxity still counted in the computing time of the second occurrence of F_1.

Accurate laxity lies between the lower bound and upper bound, and can be represented by the response time of the tasks:

$$l_{resp,i} = min_{S_k}(1 - \frac{R_k}{T_k}) \tag{4}$$

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. The laxity prediction algorithm is shown as Algorithm 1.

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. OPTIMIZATION PROCEDURE

The optimal solution mainly depends on the mapping of the synchronous sets into tasks and the priority assignment. (Natale and Pappalardo, 2008) proposed a two-stage optimization search procedure to reach the minimum type LH buffers.

The task mapping procedure will use the rate monotonic (RM) scheduling algorithm to map sets into blocks and decide the priority for each task. Every time the RM procedure begins, the algorithm will receive all the sets with no incoming links. The RM 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; if there is more than one set with the same high sampling rate, laxity prediction method will be used to decide which set should have the highest priority. As Equation 4 indicates, laxity refers to the time remaining after completing the task chain, so the smaller the laxity time is, the harder it is for that task to be scheduled. Therefore, the task set with the least laxity must have the highest priority. By applying RM algorithm and 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 apply RTW solution to evaluate the schedulability of the system by adding an RT block, and possibly a delay, on every link. All the synchronous sets with the same rate are mapped to the same task, and standard rate monotonic fixed-priority analysis is performed on the task sets. If this solution is not schedulable, there is no other fixed-priority assignment that 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.

If RTW solution is schedulable, we can test the schedulability of No Type LH mapping solution, with no type LH transaction buffers in the graph. The minimum buffer cost should be the cost of buffers in the No Type LH mapping solution. If the task mapping under this solution is schedulable, there is no room for optimization because there is no type LH buffer to remove.

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 optimized mapping solution. The breadth-first search procedure is described in (Natale and Pappalardo, 2008).

3.1 Laxity Release Optimization

As we can see from the above analysis, the laxity of one functional block indicates the schedulability of the task associated with that functional block in the system. The lower the laxity is, the more time that task has in which to finish before its deadline. As the response time calculation of laxity indicates, 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 task τ_i a *Dangerous Task* if the laxity of that functional block meets the condition $l_i < laxityGuard$, $laxityGuard \in \mathbb{R}$ and $laxityGuard \in (0, 1)$. $laxityGuard$ is a user-defined real number acting as a threshold value. If the laxity of one block is in the dangerous zone, we can use the *Laxity Release* method to increase the laxity, thus improving the schedulability of the tasks. If the worst-case computing time is fixed as an practical experimental parameter for the tasks, only the frequency time can be adjusted.

3.2 Calculation of Laxity Release Bound

For $l_{upper,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k})$, changing T_k will directly change the laxity. 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 $T_k < T_l$, $l_{lower,i} = min_{S_k}(1 - \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k})$, 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 to release the laxity is to increase 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 $\frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k^{'}} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, and the laxity of S_i is released.
  More generally, if $T_k^{'} = mT_k$, $m \in \{2^{\mathbb{N}}\}$ and still $T_k^{'} < T_l$ for every $S_l \in pre(S_k)$, $\frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k^{'}} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, and the laxity of S_i is enlarged, or *Released*.
  Notice that here, we make $m \in \{2^{\mathbb{N}}\}$ so that the system still meets the FD situation requirement after the laxity release process.
- Case 2: if $T_k^{'} = mT_k \geq T_l$ for every $S_l \in pre(S_k)$, $m \in \{2^{\mathbb{N}}\}$. Notice that $T_k < T_l$ for all the $S_l \in pre(S_k)$, in the *FD* situation, the only possibility is that $T_l = m^{'}T_k, m^{'} \in \{2^{\mathbb{N}}\}, m^{'} < m$ for every $T_l \in pre(S_k)$. $\lceil \frac{T_k^{'}}{T_l} \rceil = \lceil \frac{mT_k}{m^{'}T_k} \rceil = \frac{m}{m^{'}} > 1$ for $S_l \in pre(S_k)$, $\frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k^{'}}{T_l} \rceil C_l}{T_k^{'}} = \frac{\sum_{S_l \in pre(S_k)} \frac{m}{m^{'}} C_l + C_k}{mT_k} = \frac{\sum_{S_l \in pre(S_k)} \frac{1}{m^{'}} C_l + \frac{C_k}{m}}{T_k} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, and the 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 $T_k = T_l - \delta_k$, $\delta_k \geq 0$, $\lceil \frac{T_k^{'}}{T_l} \rceil = \lceil \frac{mT_l - m\delta_k}{T_l} \rceil \leq m$, $\frac{\sum_{S_l \in Pre(S_k)} \lceil \frac{T_k^{'}}{T_l} \rceil C_l}{T_k^{'}} \leq \frac{\sum_{S_l \in A} mC_l + \sum_{S_l \in B} C_l}{mT_k} = \frac{\sum_{S_l \in A} mC_l + \sum_{S_l \in B} \frac{C_l}{m}}{mT_k} = \frac{\sum_{S_l \in Pre(S_k)} C_l - \sum_{S_l \in B} \frac{(m-1)C_l}{m}}{T_k} < \frac{\sum_{S_l \in Pre(S_k)} C_l}{T_k}$, and the laxity of S_i is released.

In the above three cases, all the laxity of S_i can be released by increasing 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*. The tail release algorithm is shown as Algorithm 2.

Internal Release If $T_k \geq T_l$ for all the $S_l \in Pre(S_k)$, then $l_{lower,i} = min_{S_k}(1 - \sum_{S_l \in Pre(S_k)} \frac{C_l}{T_l})$. We can see that laxity will be released if we use the *Tail Release* method to increase T_k. In addition to *Tail Release*, there is another

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

```
laxity_i, S_k ← LaxityPrediction(S_i)
while laxity_i < laxityGuard do
    T_k ← 2T_k;
    UpdateGraph(S_k);
    laxity_i, S_k ← LaxityPrediction(S_i)
end while
```

way to adjust the laxity. For any $S_p \in pre(S_k)$, $T_p \le T_k$, increase T_p to $T_p' = mT_p$ and see how the laxity changes.

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

This method is called *Internal Release*. For task set S_k, we define the tense $Tense_k = \frac{C_k}{T_k}$. *Internal Release* is performed by finding the most tight working task set S_p and increasing T_p to release the laxity. If there is more than one task set with the same tense, select the task set S_p with the greatest worst-case computing time and increase T_p to release the laxity. The internal release algorithm is shown as Algorithms 3 and 4.

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 laxity_i < laxityGuard do
    S_p ← FindMostTightWorkingSet(Pre_{S_k})
    T_p ← 2T_p;
    UpdateGraph(S_k);
    laxity_i, S_k ← LaxityPrediction(S_i)
end while
```

If there $\exists S_l \in Pre(S_k)$ so that $T_k \ge T_l$, 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.

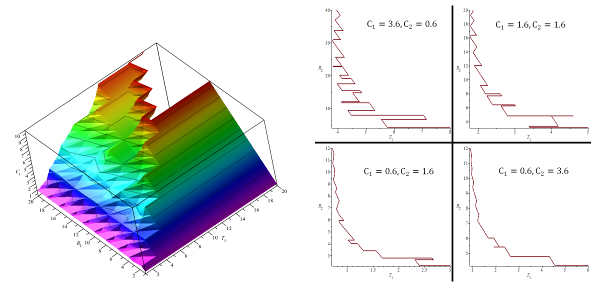
3.3 Response Time of Laxity Release

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 $R_k \le T_k$, task set S_k can finish computing before its deadline. For $R_k = \sum_{S_l \in pre(S_k)} \lceil \frac{R_k}{T_l} \rceil C_l + C_k$, the tail release method can always be applied for laxity release. We will now show how internal release can be applied to $l_{resp,i}$.

Algorithm 4 Find Most Tight Working 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 ← FindMostTightWorkingSets(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
```



(a) R_2-T_1-C_1 relation graph (b) R_2-T_1 relation graph

Fig. 1. R-T-C relation graph

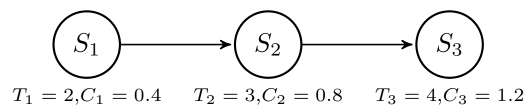


Fig. 2. Laxity Release Example

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, and C_2 is shown in Fig. 1(a). Response time is defined as the minimum nonnegative number that fulfills the equation. As shown in Fig. 1(b), for different C_1 and C_2, R_2 tends to decrease as T_1 increases. This result can be extended to the general condition $R_k = \sum_{S_i \in \{pre(S_k) - S_l\}} \lceil \frac{R_k}{T_i} \rceil + \lceil \frac{R_k}{T_l} \rceil C_l + C_k$. In general, the increase of T_l will lead to the decrease of S_k. Thus, internal release can be applied to the response time calculation of laxity.

The *Laxity Release Procedure* is defined as follows: in the priority assignment procedure, compute the laxity of every candidate task. If task S_k has a l_k so that $l_k < laxityGuard$, search in the tasks with priority greater than S_k with the maximum *tense* and increase the frequency time of that task. Repeat this procedure until every candidate task S_k fulfills $l_k \ge laxityGuard$, and assign priority using a root mean squares (RMS) calculation.

Here, we take the system in Fig. 2 as an example. Functional block 1 has a period time of $T_1 = 2$ and worst-case execution time $C_1 = 0.4$. For block 2 and 3, $T_2 = 3$, $T_3 = 4$, $C_2 = 0.8$, and $C_3 = 1.2$. The utilization factor is $U = 0.4/2 + 0.8/3 + 1.2/4 = 0.767$. By using the laxity prediction algorithm, we can find out the laxity of the three blocks, where $l_1 = 0.8$, $l_2 = 0.6$, and $l_3 = 0.3$. Then, functional block 3 needs to be optimized because $1 - l_3 > U$. We searched in blocks with higher priority than block 3, and we found block 1 with the minimum frequency time. $T_1 = 1$ will increase to $T_1 = 2$. Thus, $l_1 = 0.9$, $l_2 = 0.65$, and $l_3 = 0.833$. We can see that after laxity release optimization, no laxity is less than the total utilization of the system, which means the schedulability of the system is improved.

4. EXPERIMENTAL EXAMPLES

Table 1. Experimental Results for different util

Utilization	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85
$l_{\mathrm{upper_o}}$ buf	0	0	4	14	14	28	36	64
l_{resp} buf	0	0	4	10	12	20	26	46
release buf	0	0	4	10	12	20	22	40

Table 2. Experimental Results for util=0.85

	Test Cases												sum
$l_{\mathrm{upper_o}}$ buf	0	4	12	8	6	2	2	8	4	6	6	6	64
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 algorithms performance, we will compare the buffer cost by using different laxity prediction algorithms and the laxity release algorithm. The experimental results were 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 has 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 ×2, the base rate ×3, and the base rate ×5. Each graph set contains 12 randomly generated graphs, and the buffer optimization procedure is performed on the graphs. The type LH RT-transaction buffer cost is fixed at 2.

The parameter *laxityGuard* is a threshold value and can be adjusted to meet different practical needs. As laxity and system utilization factors both characterize the schedulability of the system from different points of view, we set $laxityGuard = 1 - U$ in the experiments, with $U = \sum_i \frac{c_i}{T_i}$ indicating the system utilization.

Table 1 shows the results of adding all the buffer costs in each graph set for different utilizations. The original laxity upper bound $l_{upper_o,i}$ from Equation 1, the response time calculation $l_{resp,i}$ algorithms, and the laxity release algorithm were used. We can see that for low CPU utilizations, all three methods generate a very low buffer cost. As the utilization grows, buffer cost grows significantly, and the algorithms proposed in this paper can generate a better result, i.e., lower buffer cost, than $l_{upper_o,i}$. When the utilization reaches 0.8 and system laxity is relatively low, the laxity release methods come into effect. Table 2 shows the detailed results of the 12 test cases in the graph set with $utilization = 0.85$. The $l_{resp,i}$ algorithm generates a better result than $l_{upper_o,i}$, and the laxity release method can result in an even lower buffer cost.

5. CONCLUSIONS

This paper analyzed the laxity prediction and release problem in the buffer optimization procedure of Simulink multitasking models. Our research demonstrates the possibility of improving the performance of laxity prediction in order to reduce the system cost of Simulink multitask implementation. Experiments were carried out with our approach of laxity prediction based on response time, and the results showed that the performance of priority assignment is improved and buffer cost is reduced. 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 dangerous tasks and release the laxity to improve the system schedulability and reduce the system buffer cost.

REFERENCES

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

Astrom, K. and Wittenmark, B. (2011). *Computer-controlled systems: theory and design.* Dover Publications.

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

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

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

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

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

Natale, M. and Pappalardo, V. (2008). Buffer optimization in multitask implementations of simulink models. *ACM Transactions on Embedded Computing Systems (TEC-S)*, 7(3), 23.

Scaife, N. and Caspi, P. (2004). 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*, 119–126. IEEE.

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

Tripakis, S., Sofronis, C., Scaife, N., and Caspi, P. (2005). 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*, 353–360. ACM.