Towards a Practical Implementation of Criticality Mode Change in RTOS

Young-Seung Kim  
Hyun-Wook Jin  
Department of Computer Science and Engineering, Konkuk University  
Seoul 143-701, Korea  
\{truth50, jinh\}@konkuk.ac.kr

Abstract

In order to address the trade-off between certification and resource efficiency, researchers are recently trying to apply a criticality mode change mechanism to mixed-criticality systems. However, the actual implementation of the criticality mode change has not been studied rigorously. In this paper, we suggest a practical design to implement the criticality mode change framework for Real-Time Operating Systems (RTOS). In particular, we aim to minimize the scheduler overheads while maximizing the resource efficiency. To the best of our knowledge, this is the first work in the literature that presents an actual implementation of the criticality mode change in RTOS.  

1 Introduction

In modern embedded systems, such as avionics and automotive systems, several applications that have different criticality levels are consolidated on a single computing node aiming for small size, light weight, and low power [6]. However, the over-provisioning of computing resources or a pessimistic schedulability test for the certification causes a serious underutilization of resources.

In order to address the trade-off between certification and resource efficiency, researchers are recently trying to apply a criticality mode change mechanism to the task scheduling [11, 4, 2]. For instance, in a dual-criticality system, tasks have two Worst-Case Execution Times (WCETs) for low (LO)- and high (HI)-criticality modes, respectively, where one is suggested by the system designer for resource efficiency, while the other is given with more pessimism for safety-critical certification. The system starts in the LO-criticality mode, where the tasks run assuming the WCET given by the system designer, and if a task with the HI criticality experiences an overrun, it enters the HI-criticality mode to ensure safety critical functions. Since the WCET for the HI-criticality mode is larger, we may not be able to run all tasks with real-time guarantee in this mode; thus, tasks with the LO criticality may be dropped.

Although there are several implementation studies for mixed-criticality systems, the practical implementation of the criticality mode change has not been given enough attention. Herman et al. [7] and Anderson et al. [1] presented implementations of the mixed-criticality scheduling framework on a modified Linux, but the criticality mode change was not considered in their implementations. An implementation of the criticality mode change was presented by Baruah and Burns [3], but the detailed design and the performance evaluation were not provided. Huang et al. [8] implemented several mixed-criticality scheduling algorithms on Linux and compared these. This study focused on the on-line scheduling; thus, the additional operations to handle the criticality mode change are involved at run-time and increase the system overheads.

In this paper, we suggest a practical design to implement the criticality mode change framework in Real-Time Operating Systems (RTOS). We aim to minimize the additional scheduler overheads while maximizing the resource efficiency. In this study, we assume that the priority of tasks are given by an off-line scheduler. Our preliminary implementation especially targets eCos (embedded Configurable operating system) [10], a free and open source RTOS. The performance evaluation shows that our extension adds very low overheads.

2 System Model

We use the dual-criticality periodic task model that defines a task $\tau_i = (p_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where $p_i$ is the period and $\chi_i \in \{\text{LO, HI}\}$ is the criticality of the task. Tasks with the HI criticality (i.e., $\chi_i = \text{HI}$) have to be certified, while LO criticality tasks (i.e., $\chi_i = \text{LO}$) do not. A task with the HI criticality has two WCETs denoted as $C_i(\text{LO})$ and $C_i(\text{HI})$. $C_i(\text{LO})$ is given by the system designer with less pessimism aiming for high resource utilization. On the other hand, $C_i(\text{HI})$ is decided for the safety-critical certification; thus, $C_i(\text{LO}) < C_i(\text{HI})$. In the case of the LO criticality task, $C_i(\text{LO}) = C_i(\text{HI})$. We assume that all tasks arrive at the same time and the deadline is the same with the end of its period, but our design and implementation described in Section 3 does not rely on this assumption.

Each task gives rise to a cyclic sequence of jobs. $J_i^k$
denotes the \(k^{th}\) job of \(\tau_i\), and \(P(J^k_i)\) specifies the priority of \(J^k_i\). We assume that an off-line scheduler decides priority of jobs for the least common multiple of periods (i.e., hyper-period). The off-line scheduler can assign a priority to each job regardless whether jobs belong to the same task or not [2].

The criticality mode of a task is changed from LO to HI when the task \(\tau_i\) overruns (i.e., exceeds its execution budget) consecutively more than the threshold, which can also be extended for \((m,k)\)-model [5]. The intention behind this is to suppress the mode change as much as possible because the HI-criticality mode may result in the under-utilization of resources. In fact, an overrun does not immediately cause a hazardous result in many applications. For example, aerial vehicles do not easily crash by an occasional overrun of a flight control task thanks to the lift, but the consecutive overruns put the vehicle in danger. In addition, each task manages a separate criticality mode, but the consecutive overruns put the vehicle in danger. In occasional overrun of a flight control task thanks to the lift, immediately cause a hazardous result in many applications. Utilization of resources. In fact, an overrun does not implement the suggested design without significant additional scheduling overheads.

### 3 Criticality Mode Change Framework

In this section, we detail our design and implementation of the criticality mode change for RTOS. By extending the data structures and scheduling algorithm of an existing RTOS, we can easily implement the suggested design without significant additional scheduling overheads.

#### 3.1 Task Control Blocks

Figure 1 shows the kernel data structure called task control block, which consists of the task’s period \((p_i)\), criticality \((\chi_i)\), WCETs \((C_i(LO)\) and \(C_i(HI)\)), overrun threshold, current mode \((\mu_i)\), runtime in the current period, and number of overruns. The current mode represents the criticality mode of the task, and is updated by the task scheduler described in the next subsection. The runtime is increased also by the task scheduler and used to detect an overrun. This value is reset to zero at the start of a new period. The number of overruns is increased when the task experiences an overrun consecutively and is reset to zero once the deadline is met. If the number of consecutive overruns becomes larger than the overrun threshold of the task, its criticality mode is raised.

The task control block also has a circularly linked-list to specify the priorities \((P(J^k_i))\) of the jobs for hyper-period. As mentioned in Section 2, we assume that the priorities are provided by an off-line scheduler. Whenever a new period starts, the task is given the priority of the next job in the job list. Actual scheduling of the task is performed by the fixed-priority scheduler described in the next subsection.

As we have described in Section 2, a task changes its criticality mode independently from that of other tasks aiming for resource efficiency. That is, if a LO criticality task is still schedulable even after a HI criticality task changes its mode to HI-criticality, we allow the LO criticality task to run. Since we assume the off-line schedulers, we do not consider the schedulability test at runtime, while the on-line schedulers have to perform this whenever a mode change happens. Thus, we can avoid the schedulability test overhead during run-time, but then, the tricky issue is how to efficiently represent the interference relationships in the task control blocks.

If a job \(J^k_i\) of the task \(\tau_i\) cannot be scheduled when the task \(\tau_j\) enters the HI-criticality mode (i.e., \(\mu_j = HI\)), we say that \(J^k_i\) has an interference relationship with \(\tau_j\) and represent this as \(\tau_j \rightarrow J^k_i\), otherwise, \(\tau_j \nrightarrow J^k_i\). In addition, if \(J^k_i\) does not have an interference relationship with a set of tasks, \(\tau = \{\tau_j, \ldots\}\), we represent this as \(\tau \nrightarrow J^k_i\).

Since we target dual-mode systems, an interference relationship of a job with other tasks can be embodied in a bitmask such that \(\text{bitmap}^k_i[j] = 1\), \(\forall j\) that \(\tau_j \in \tau\). Thus, if there are \(n\) ones in the bitmask, it can present \(2^n - 1\) combinations of other tasks’ mode that allow the corresponding job to be scheduled. For example, if \(\{\tau_a, \tau_b\} \rightarrow J^k_i\), \(\text{bitmap}^k_i[a] = 1\) and \(\text{bitmap}^k_i[b] = 1\). Since we may need several bitmasks to represent all possible cases, we allow the system designer to limit the number of bitmasks in our implementation. We discuss detail algorithm to utilize this bitmask in Sections 3.2 and 3.3.

#### 3.2 Task Scheduler

Most of existing RTOS perform the tick-based task scheduling that regularly generates a timer event and invokes the task scheduler. Our design is based on the tick-based scheduler and can be easily implemented by modifying an existing scheduler as shown in Algorithm 1.

The task scheduler first increases the runtime of the current task \(\tau_out\) to track how long the task has occupied the processor resources (line 1). Then the scheduler evaluates if the task overruns (line 2). If the overrun task is a LO criticality one, the scheduler simply stops the current job and considers the next job in the next period (lines 3-4). If a HI criticality task misses its deadline, the scheduler increases the number of consecutive overruns (lines 5-6). If the the number of consecutive overruns reaches the threshold and the current mode is LO (line 7), the scheduler changes its criticality mode to HI (line 8) and indicates this into the bitmap, \(\text{bitmap}_{mode}\), that represents the criti-


Algorithm 1 Algorithm for task scheduler

1: increase the runtime of τ_i  
2: if runtime_i > C_i \( \times \) (X_i) then  
3: if \( X_i == \text{LO} \) then  
4: skip to the next job  
5: else if \( X_i == \text{HI} \) then  
6: overruns ← overruns + 1  
7: if overruns == threshold and \( \mu_i == \text{LO} \) then  
8: \( \mu_i \leftarrow \text{HI} \)  
9: bitmap_mode[i] \( \leftarrow \) 1  
10: else if overruns == threshold and \( \mu_i == \text{HI} \) then  
11: alert the user to a fatal system error  
12: else  
13: skip to the next job  
14: end if  
15: end if  
16: end if  
17: for tasks that start a new period do  
18: if bitmap_mode[\( i \)] = bitmap_mode[\( k \)] then  
19: change \( \tau_i \)'s priority to \( P(\tau_i) \)  
20: move \( \tau_i \) to the ready queue  
21: end if  
22: end for  
23: next ← PickNext()  
24: ContextSwitch(current, next)

To depict how the detailed information is stored in the task control blocks, we use an example presented in [2], where the author targets synchronous reactive tasks. Table 1 shows the task set. The job priorities of the given tasks are defined as \( J_1 \gg J_2 \gg J_3 \gg J_4 \), by the OCBP priority ordering suggested by Baruah [4].

Figure 2 shows the task control blocks of \( \tau_1 \) and \( \tau_5 \). We assume that there are two bitmasks for each job to represent the interference relationships. Since the hyper-period of these tasks is 12, only one job is in the job list of \( \tau_1 \), while two jobs are specified in \( \tau_5 \)'s. We can see that the current mode of \( \tau_1 \) is HI and thus the \( \text{bitmap}_{\text{mode}} \) is 10000. The first job of \( \tau_5 \) has bitmasks 10000 and 01000 because \( J_3 \) is still schedulable when either \( \tau_1 \) or \( \tau_2 \) enters the HI-criticality mode. Thus, in this situation, \( J_3 \) is runnable because \( \text{bitmap}_{\text{mode}}(10000) \gg \text{bitmap}_{\text{mode}}(10000) = 10000 \) by the line 18 of Algorithm 1. \( J_2 \) is also schedulable even if \( \tau_2 \) enters the HI-criticality mode, but we cannot represent this case because we assume only two bitmasks per job in this example. On the other hand, \( J_2 \) can be schedulable regardless of the current mode of HI criticality tasks (i.e., \( \tau_1, \tau_2, \) and \( \tau_4 \)). Therefore, all bitmasks of \( J_2 \) are 11111.

4 Evaluation

We implemented the suggested design in eCos (ver. 3.0) and analyzed its overheads on an industrial embedded board equipped with an Intel i3 2.2GHz processor. To measure the overheads of the task scheduler, we generated timestamps from the functions for mode change (lines 5-
9 in Algorithm 1), periodic release (lines 17-22 in Algorithm 1), finding the next task (line 23 in Algorithm 1), and context switch (line 24 in Algorithm 1) by inserting the rdtsc instructions into the eCos kernel.

Figure 3 shows the ratio of the task scheduling overheads with a single task. Our implementation adds the mode change and periodic release overheads, which are less than 360 ns in the worst case. Figure 4 shows the cumulative probability of the periodic release and mode change overheads. As we can see, the mode change operation adds a very little overhead and shows significantly less jitter than the periodic release operation. The periodic release overhead increases in proportional to the number of tasks as shown in Figure 5. In addition, we observed that the overhead to skip to the next job (lines 4 and 13 in Algorithm 1) is about 3µs.

5 Conclusions and Future Work

In this paper, we suggested a practical design to implement the criticality mode change framework. In order to mitigate the inflexibility of the off-line schedulers and improve the resource efficiency, the framework provides the bitmasks that represent the interference relationships between tasks. We implemented the suggested design in the eCos kernel and showed that our implementation added low overheads. As future work, we need to study an algorithm for providing the optimal bitmasks of interference relationship. We also plan to extend our design and implementation for a hierarchical scheduler.

References