Dynamic Priority Inversion Avoidance in Real-Time Operating Systems

I-Hsuan Huang, Chun-Yen Yu, Yu-Chan Chung, and Cheng-Zen Yang
Department of Computer Science and Engineering, Yuan Ze University, Taiwan, R.O.C.
{ihhuang, czyang}@syslab.cse.yzu.edu.tw
{s941522, s941520}@mail.yzu.edu.tw

Abstract—One of the most important design goals in a real-time system is to guarantee that all tasks can finish their work before reaching deadline. However, the sequence of resource allocation among tasks of different priorities may result in a severe priority inversion problem such that a high priority task will violate its deadline requirement. Although previous solutions can shorten the latency resulted from the inversion, they do not address how to avoid the occurrence of priority inversion. In this paper, we propose a dynamic priority inversion avoidance mechanism to avoid the priority inversion. According to the simulation results, the proposed mechanism can guarantee the real-time requirements of the real-time system.

Keywords: Priority Inversion Avoidance, RTOS, Resource Allocation

1. Introduction

In embedded system design, real-time considerations play an important role to the success of time-critical applications in many areas. Example applications include aerospace control systems, vehicle transportation systems, and health-care appliances [10]. To process numerous activities in real-time systems to meet their timing constraints, a major design consideration in system design is task scheduling [10]. With a real-time task scheduling algorithm, the system is guaranteed to have as many real-time tasks be completed before the deadlines as possible. However, for tasks sharing some critical resources, a severe problem may arise to suppress the effectiveness of the scheduling algorithm when the priority inversion phenomenon occurs [11]. During the inversion period, low-priority tasks hold the critical resources, and the high-priority tasks sharing the same critical resources must wait until the low-priority tasks release the resources. Therefore, the completion times of the suspended high-priority tasks are lengthened, and they may further exceed the deadline requirements.

Traditionally, several resource access protocols have been proposed to mitigate the influences of priority inversion. For example, the priority inheritance protocol (PIP) can dynamically and temporarily change the priority of some tasks such that the suspension delay is minimized [11]. Another famous protocol is the priority ceiling protocol that can avoid deadlocks and chained blocking [5]. Although these priority-changing protocols can effectively reduce the maximum delay of the suspended high-priority tasks for many real-time applications, the suspended high-priority tasks must still wait and may fail the timing constraints.

In [1], a different approach is proposed to avoid priority inversion. Instead of temporarily raising the priority of the resource-holding tasks, the operating system adjusts the task execution sequence to prevent the occurrences of priority inversion before task execution. Through the statically re-arranged resource allocation sequence, the adjustment can effectively guarantee that high-priority tasks complete their work before the deadlines. Nonetheless, the rearrangement proposed in [1] requires that the operating system can get prior information of task execution. For a real-time system with dynamically issued tasks, the static rearrangement scheme may still get mis-prediction and cannot effectively prevent the occurrence of priority inversion.

In this paper, we propose a dynamic priority inversion avoidance mechanism for real-time operating systems (RTOS). The proposed mechanism employs a window-based prediction scheme to help the RTOS learn the historical execution behavior of real-time tasks. If some task has a similar execution pattern in a period of time, the RTOS will learn the pattern and dynamically rearrange the task execution sequence according to the collected information to prevent potential priority inversion by deferring lower-priority tasks sharing the same critical resource. For tasks which cannot be deferred due to the deadline requirements, the RTOS adopts the essential PIP scheme to avoid worsening the priority inversion phenomenon. Based on the window-based prediction scheme, the RTOS can guarantee more high-priority tasks to complete their work without the interference of priority inversion. Therefore, the deadline-miss rate can be effectively reduced. In our simulation to study the performance effectiveness, the results are positively show that the mechanism can actually reduce the number of deadline misses.

The rest of this paper is organized as follows. Section 2 gives a brief review to the related work on the priority inversion problem. Section 3 describes the system model.
In Section 4, the dynamic priority inversion avoidance mechanism is elaborated. Finally, Section 5 demonstrates the simulation results and Section 6 concludes the paper.

2. Related work

Priority inversion has been considered as an important problem for real-time task scheduling [1][2][16]. When priority inversion occurs, it is very highly possible for suspended high-priority tasks to miss the deadline requirements. To alleviate the severe latency problem introduced by priority inversion, numerous approaches have been proposed and can be mainly classified into two categories: temporarily priority-changing approach and execution rearrangement approach.

In the temporarily priority changing approach, the priority of the lower-priority task which holds the sharing critical resource is temporarily raised to avoid unexpected lengthy chained blocking due to priority inversion. Several resource access protocols of this approach, such as the priority inheritance protocol (PIP) [8][12][13] and the priority ceiling protocol (PCP) [5], have been proposed and shown their effectiveness in many practical real-time applications. With PIP, RTOS optimistically let all tasks be executed without any delay when they arrive [7][8]. If the priority inversion phenomenon occurs, the RTOS immediately raises the priority of the lower-priority task which holds the shared critical resource to the highest priority to ensure that the resource can be released as soon as possible. Hence, the suspension of the waiting high-priority task is minimized as the period of executing only the priority-raised task. PCP is an alternative to PIP [8][14][15] and the priority ceiling protocol (PCP) [5], have been proposed and shown their effectiveness in many practical real-time applications. With PIP, RTOS optimistically let all tasks be executed without any delay when they arrive [7][8]. If the priority inversion phenomenon occurs, the RTOS immediately raises the priority of the lower-priority task which holds the shared critical resource to the highest priority to ensure that the resource can be released as soon as possible. Hence, the suspension of the waiting high-priority task is minimized as the period of executing only the priority-raised task. PCP is an alternative to PIP [8][14][15].

In PCP, the RTOS defines a ceiling priority for each critical resource. Then the priority of each task acquiring the critical resource is temporarily raised to the ceiling priority to guarantee to have the shorted execution period. Therefore, the waiting latency is minimized. With these temporarily priority changing approaches, the maximum delay of suspended higher-priority tasks is effectively minimized when priority inversion occurs. However, they do not guarantee that the suspended higher-priority tasks can be completed before the deadline. Hence, some poor-designed lower-priority tasks may have the chance to spoil the real-time requirements of higher-priority tasks. In addition, as long as that the execution time of the lower-priority task may be varied according to the runtime data, the maximum delay of the suspended higher-priority tasks is unpredictable. The unpredictable maximum delay may thus complicate the design of the real-time system.

In the execution rearrangement approach, the execution characteristics of each task need to be first collected and the resource request information is analyzed accordingly. The task scheduler will adjust the task execution sequence based on the resource request information to avoid the occurrence of priority inversion. In [1], Babaoğlu, Marzullo, and Schneider provide a formal analysis of priority inversion and show a set of sufficient conditions for preventing its occurrence. If a system designer performs task execution analysis in the design phase, priority inversion can be prevented based on the detection of these sufficient conditions priori. However, this prevention approach is only suitable for a statically scheduled real-time system in which the knowledge of task execution is all available in the design phase. In [3][4], an anomaly is further reported for this approach when task execution characteristics are changed due to system environment alternations, such as hardware upgrade. Accordingly, some anomaly prevention rules are proposed to prevent priority inversion by rearranging the task execution order. However, these anomaly rules only consider real-time systems in which task execution characteristics can be precisely acquired priori. For real-time systems in which task execution characteristics cannot precisely predicted or can be only imprecisely predicted, the anomaly prevention rules will still suffer.

In fact, the task execution behavior in the run time is usually different from the predicted behavior in the design phase. Additionally, many real-time systems have dynamic tasks whose executions depend on external events which are unpredictable in the design phase. A static rule-based approach may get corrupted in such situation. In contrast, a dynamic mechanism may be more effective by learning the task execution characteristics in the runtime.

3. System model

The system considered in this paper is a preemptive and priority-based real-time embedded system, which has an RTOS. All tasks are scheduled according to a priority-based scheduling algorithm. In the design phase, each task is associated to a fixed priority number.

This system has three kinds of tasks to run: periodic tasks, aperiodic tasks, and sporadic tasks. As defined in [6], the definitions of these tasks are similarly defined as follows:

- **Periodic tasks**
  A periodic task is executed repeatedly after the system starts up. Each periodic task is characterized by its worst-case execution time (WCET), the period, and the deadline. We assume that each periodic task must complete its work before reaching the next period.

- **Aperiodic tasks**
  An aperiodic task has its WCET and the deadline, but does not have a specific period. Therefore, it may run only once in the system, or randomly run several times. The example tasks include the system initialization procedure and the shutdown procedure. For simplicity, we assume each aperiodic task must complete its job before next task arriving and it runs only once in this paper.

- **Sporadic tasks**
  A sporadic task is service routines triggered by the external interrupt sources. Since the occurrence of ex-
ternal interrupt is unpredictable, the sporadic task may delay the original schedule and cause tasks missing deadline. To characterize the sporadic task, each sporadic task has the WCET, the minimum inter-arrival time ($\lambda$), and the deadline. We assume each sporadic must complete its work before the next interrupt signal comes.

In this paper, we use $S = \{t_1, ..., t_n\}$ to represent the task set $S$ of totally $n$ tasks in the system. Each periodic and aperiodic task has been assigned a fixed priority number ranging from 1 to $k$. The smaller priority number represents the higher priority. Each task is defined by $(r, C, D, T, \lambda)$, where $r$ is the task release time, $C$ is the WCET, $D$ is the task relative deadline, $T$ is the task period, and $\lambda$ is the minimum inter-arrival time. The number of each sporadic task is named according to its external interrupt source number. When a task completes its job or releases the CPU ownership, the RTOS picks the next task based on the priority-based scheduling algorithm. The context switch overhead is ignored.

Whenever the RTOS executes a new task, it utilizes the proposed priority inversion avoidance mechanism to guarantee that high priority tasks can finish jobs before reaching deadline. If the operating system invokes a sporadic task due to external interrupt, the original schedule is suspended and postponed. If the low priority task is delayed due to executing sporadic task, the following high priority task is chain-delayed. If the high priority task is arrived, but the CPU is still running the delayed low priority task, the phenomenon is defined as the priority inversion. Meanwhile, if the sporadic task runs for a long period, the delayed task may miss the deadline.

In this study, to avoid tasks missing deadline due to priority inversion phenomenon, we define the problem as follows. For a real-time system with priority-based scheduling, when executing a new task, the RTOS utilizes the historical execution behaviors of real-time tasks like the WCET, the period, and the deadline to avoid the priority inversion phenomenon and to minimize the amount of task missing deadlines.

### 4. Priority inversion avoidance algorithm

The problem we observed in the study is that current real-time system usually adopts the priority-based scheduling algorithm to guarantee real-time tasks meeting their deadlines. However, the priority inversion phenomenon makes the timing of tasks being unpredictable. The consequence is that tasks may violate the timing constrains.

To make sure the high priority tasks not missing deadline due to priority inversion phenomenon, in the proposed mechanism, whenever a task is executed, the real-time operating system first calculates the historical WCET and the historical execution period of the executed task and the consequent tasks. Meanwhile, all scheduled tasks are delayed. Then the real-time operating system evaluates the completion time of the consequent tasks to make sure that no high priority tasks missing deadlines. If any high priority task is scheduled after the low priority task and the high priority task may miss deadline due to priority inversion phenomenon, the real-time operating system dynamically rearranges the execution sequence of the high priority task and the low priority task.

To illustrate the concept of the proposed mechanism, as shown in Figure 1, Figure 1(a) shows two periodic tasks, the high priority task $t_1$ and the low priority task $t_2$. The actual execution time of $t_1$ is one time unit and the actual execution time of $t_2$ is two time units. The periods of $t_1$ and $t_2$ are both three time units. The deadlines of $t_1$ and $t_2$ are same with the next respective task arrival time. Each task holds the critical resource when one request is arrived and releases the critical resource when the request is completed.

Demonstrated in Figure 1(b), an external interrupt comes at time $t + 2$ and triggers the sporadic task $t_3$. The actual execution time of $t_3$ is 3 time units. Meanwhile, $t_2$ is interrupted and is executed. To simplify the case, we assume the collected historical WCET and period of the three tasks are equal to their real behaviors. At time $t + 5$, $t_3$ finishes its job. Before the operating system resumes $t_2$, the operating system evaluates the laxities of $t_1$ and $t_2$. According to the prediction, $t_1$ arrivals at time $t + 3$ and the corresponding deadline is at time $t + 6$, $t_2$ arrivals at time $t + 4$ and the corresponding deadline is at time $t + 7$. The laxities of $t_1$ and $t_2$ are 1 and 2 time units respectively. At time $t + 5$, if the operating system directly resumes $t_2$, $t_1$ may miss the deadline. Utilizing the proposed mechanism, the operating system expects that the complete time of $t_2$ is at time $t + 6$ and the complete time of $t_1$ is at time $t + 7$. Hence, the operating system considers that $t_1$ may miss its deadline and identifies the situation as the priority inversion phenomenon. Then, the operating system delays $t_2$ until $t_1$ finishing its job. In this case, if the system is still schedulable after running sporadic tasks, both the task $t_1$ and the task $t_2$ fulfill their timing constrain.

Figure 1(c) presents the case that the schedulability cannot be fulfilled. If the sporadic task $t_3$ runs 1 time unit at time $t + 1$ and $t + 4$ respectively, either $t_1$ or $t_2$ may miss deadline at time $t + 5$. Since the task $t_1$ has higher priority than the task $t_2$, the proposed mechanism chooses to sacrifice $t_2$ to guarantee fulfilling the timing constrain of $t_1$.

To obtain more accurate historical execution behaviors of real-time tasks, we adopt a window-based model to collect task execution information. The window size is the number of past execution records being counted. The basic idea is that the execution behaviors of real-time tasks in the window region are stable. Hence, the historical WCET and period collected in the window region can present the execution behaviors of tasks in the moment. If the deadline misses
decreases, the window size increases. If the deadline misses increases, the condition means that some task behaviors change. Meanwhile, the window size shrinks to the half of the current size to get more recent execution behaviors of tasks.

Besides the window-based model, we also apply the exponential prediction model in the proposed mechanism. The exponential prediction model means that the historical WCET and period are the exponential average of the past records. The predicted WCET of the task \( \tau_n \) is the average of the last execution time of \( \tau_n \) and the last predicted WCET of \( \tau_n \).

5. Simulation results

We developed a simulator to evaluate the proposed priority inversion avoidance mechanism. Two conditions are simulated separately. The first condition is that the priority inversion phenomenon is caused by executing sporadic tasks. We randomly generated three categories of tasks and applied the proposed mechanism to measure the amount of deadline misses. The three categories are the aperiodic task, the periodic task, and the sporadic task. The aperiodic tasks include the system startup task and the system shutdown task. The periodic tasks include the hard real-time task and the soft real-time task. The sporadic tasks are the interrupt service routines. The characteristics of these tasks are listed in Table 1.

In the simulator, during executing the system startup task and the system shutdown task, the interrupt is disabled. Thus, the sporadic tasks only appear when the real-time operating system executing the periodic tasks. The actual execution time of these tasks is randomly generated to simulate the runtime variation. The value of the execution time of each task is bounded by the defined WCET (worst-case execution time) and the defined BCET (best-case execution time). The actual interval of the sporadic tasks is bounded by the minimum interval and the maximum interval. Besides, the deadline of each task is defined according to the system model introduced in Section 3.

To evaluate the impact of the proposed mechanism, we randomly generated 10 cases to measure the amount of deadline misses. Each case includes 1 aperiodic task, 1 high priority periodic task, 1 low priority periodic task, and 1 sporadic task. Each task except the sporadic task is assigned with a fixed priority. All tasks are scheduled according to the priority numbers. Without the priority inversion avoidance mechanism, we totally observed 6 deadline misses reported by the high priority periodic task. Considering all schedulable cases, all deadline misses reported by high priority periodic task are eliminated.

For example, in one of the generated case, the task set \( S = \{ \tau_1, \tau_2, \tau_3 \} \), where \( \tau_1 \) is an aperiodic task, \( \tau_2 \) is the high priority periodic task, \( \tau_2 \) is low priority periodic task, and \( \tau_4 \) is the sporadic task. The \( \tau_2 \) and \( \tau_3 \) totally run two cycles.

Table 1: The characteristics of the tasks in the experiment.

<table>
<thead>
<tr>
<th>WCET_{aperiodic}</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCET_{aperiodic}</td>
<td>15</td>
</tr>
<tr>
<td>Deadline_{aperiodic}</td>
<td>25</td>
</tr>
<tr>
<td>WCET_{periodic}</td>
<td>25</td>
</tr>
<tr>
<td>BCET_{periodic}</td>
<td>15</td>
</tr>
<tr>
<td>Period_{periodic}</td>
<td>25</td>
</tr>
<tr>
<td>Deadline_{periodic}</td>
<td>25</td>
</tr>
<tr>
<td>WCET_{sporadic}</td>
<td>10</td>
</tr>
<tr>
<td>BCET_{sporadic}</td>
<td>8</td>
</tr>
<tr>
<td>Min_Interval_{sporadic}</td>
<td>50</td>
</tr>
<tr>
<td>Max_Interval_{sporadic}</td>
<td>100</td>
</tr>
<tr>
<td>Deadline_{sporadic}</td>
<td>10</td>
</tr>
</tbody>
</table>
Illustrated in Figure 2, without priority inversion avoidance mechanism, \( \tau_4 \) is executed at time 52 and causes a 10 time unit delays. Accordingly, after executing \( \tau_3 \), \( \tau_2 \) misses its deadline at time 86. With the proposed priority inversion avoidance mechanism, the operating system switches the \( \tau_2 \) and \( \tau_3 \) to guarantee the timing constrain of \( \tau_2 \).

The second condition is that the priority inversion phenomenon is caused by the runtime execution behavior variation. We randomly generate 1 high priority periodic task and 1 low priority periodic task. The WCET of the two tasks is bounded to 10 time units. The simulation runs 20000 time units totally. Each task is executed 3000 – 10000 times in the simulation periodically.

We compare the deadline misses of the proposed mechanism (PIA) and the well-known priority inheritance protocol (PIP) approach. Shown in Figure 3, the average deadline misses of high priority task are 30.30 and 12.90 in 10 runs. Accordingly the proposed priority inversion avoidance mechanism can significantly reduce the deadline miss rates.

6. Conclusions

Timing constrain is an important design issue in real-time task scheduling. The priority inversion phenomenon complicates the timing control. Hence, in the study, we propose a dynamic priority inversion avoidance mechanism to guarantee that high priority tasks can meet their timing constrains. The proposed mechanism relies on the collected historical execution behaviors of real-time tasks to rearrange task execution order to avoid the priority inversion. We also utilize the window-based and the exponential prediction model in the proposed mechanism. According to the simulations, the proposed mechanism can effectively prevent high priority tasks missing deadline.

7. Acknowledgment

The authors would like to thank the National Science Council of the Republic of China, Taiwan for partially supporting this research under grant NSC97-2218-E-155-005.

References


