Research Article
Assigning Priorities for Fixed Priority Preemption Threshold Scheduling

Saehwa Kim

Department of Information Communications Engineering, Hankuk University of Foreign Studies, Yongin-si, Gyeonggi-do 449-791, Republic of Korea

Correspondence should be addressed to Saehwa Kim; ksaehwa@hufs.ac.kr

Received 1 September 2015; Accepted 15 October 2015

Academic Editor: Marko Bertogna

Copyright © 2015 Saehwa Kim. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preemption threshold scheduling (PTS) enhances real-time schedulability by controlling preemptiveness of tasks. This benefit of PTS highly depends on a proper algorithm that assigns each task feasible scheduling attributes, which are priority and preemption threshold. Due to the existence of an efficient optimal preemption threshold assignment algorithm that works with fully assigned priority orderings, we need an optimal priority assignment algorithm for PTS. This paper analyzes the inefficiency or nonoptimality of the previously proposed optimal priority assignment algorithms for PTS. We develop theorems for exhaustively but safely pruning infeasible priority orderings while assigning priorities to tasks for PTS. Based on the developed theorems, we correct the previously proposed optimal priority assignment algorithm for PTS. We also propose a performance improved optimal priority assignment algorithm for PTS proving its optimality. The empirical evaluation results clearly show the effectiveness of the proposed algorithm.

1. Introduction

Preemption threshold scheduling (PTS) is an extension of preemptive fixed priority scheduling where each task has an extra scheduling attribute, called a preemption threshold, in addition to a priority. The preemption threshold of a task is its run-time priority, which is maintained after the task is dispatched and until it terminates its execution, so it regulates the degree of “preemptiveness” in fixed priority scheduling [1]. If the threshold of each task is the same as its original priority, then PTS is equivalent to fully preemptive fixed priority scheduling (FPS), and if each task has the highest threshold value in a system, it is equivalent to nonpreemptive scheduling (NPS). The use of PTS is very effective in system tuning processes since it enhances real-time schedulability, eliminates unnecessary preemptions, reduces memory stack usage [2] via the notion of nonpreemption groups [3], and allows for scalable real-time system design [4–6]. Preemption thresholds and nonpreemption groups are also parts of OSEK [7] and AUTOSAR [8] standards of automotive operating systems. As remarked in [9], PTS represents an example of a great success of transferring academic research results to industrial applications [10–12].

The benefit of enhanced real-time schedulability of PTS highly depends on a proper algorithm that assigns each task feasible scheduling attributes, which are priority and preemption threshold. The work of this paper has been highly motivated by our previous work of SISAtime [13], which adopts PTS to schedule active (concurrent) objects of real-time object-oriented models [14–19]. While SISAtime contains an optimal scheduling attributes assignment algorithm for PTS, it is not so much efficient. A scheduling attributes assignment algorithm is optimal if it is guaranteed to output a feasible (schedulable) scheduling attributes assignment if one exists [1, 20–22].

There are two previously proposed optimal scheduling attributes assignment algorithms for PTS: TRAVERSE() of SISAtime [13] and SEARCH() of [1], which is the first academic article that presented PTS. Both algorithms work in two stages. At the first stage, priorities are assigned to all tasks. At the second stage, preemption thresholds are assigned using the optimal preemption threshold assignment algorithm, OPT-ASSIGN-THRESHOLD() of [1], which has the complexity of $O(n^2)$. Recently, [23] extends OPT-ASSIGN-THRESHOLD() by considering the cache-related preemption delay (CRPD), and it also assigns optimal preemption
thresholds for tasks with preassigned priorities. Since preemption thresholds are wholly assigned at the second stage, both algorithms focus on how to assign priorities to tasks. With this, we call these optimal "scheduling attributes" assignment algorithms for PTS as optimal "priority" assignment algorithms for PTS.

In this paper, we analytically show that TRAVERSE() is inefficient and SEARCH() is not optimal. We develop theorems for exhaustively pruning infeasible priority orderings without harming the optimality of priority assignment algorithms for PTS. Specifically, we develop following lemmas and theorems under PTS:

(i) Under PTS, if the priority of a task is fixed, its worst-case response time does not decrease when its preemption threshold is lowered (Lemma 3).

(ii) Under PTS, if the preemption threshold of a task is fixed, its worst-case response time does not decrease when its priority is lowered (Theorem 6).

(iii) Under PTS, if a task with the highest preemption threshold in a priority ordering is infeasible, the task set with the priority ordering is also infeasible (Theorem 4).

(iv) Under PTS, if a task with the highest preemption threshold in a priority ordering is infeasible, the task set with another priority ordering that assigns the task the lowered priority is also infeasible (Theorem 7).

By applying these theorems, we correct SEARCH() and propose CORRECTED-SEARCH() which is more efficient than TRAVERSE(). We also propose PRUNED-TRAVERSE() that improves the performance of CORRECTED-SEARCH() and proves its optimality.

We also empirically evaluate the performances of the discussed optimal priority assignment algorithms. We first empirically show the usefulness of the proposed optimal priority assignment algorithm by showing that they always achieve the better schedulability than any other existing nonoptimal priority assignment algorithms. We also compare the actual runtimes for executing each optimal priority assignment algorithm as well as PA-DMMPT() by [24], which is the most effective heuristic priority assignment algorithm for PTS if it is combined with the policy of deadline monotonic priority ordering (DMPO). The empirical results clearly show that the actual runtimes of TRAVERSE() are reduced by CORRECTED-SEARCH(), whose actual runtimes are also more reduced by PRUNED-TRAVERSE() while such performance improvements become drastically large as the number of tasks increases. It is also shown that the actual runtimes of PRUNED-TRAVERSE() are even smaller than those of PA-DMMPT().

The remainder of the paper is composed as follows. Section 2 gives the task model with notations and presents a walk-through example that motivates our work. Section 3 analyzes previously proposed optimal priority assignment algorithms for PTS. Section 4 corrects previously proposed SEARCH() algorithm making it an optimal priority assignment algorithm for PTS. Section 5 describes our proposed optimal priority assignment algorithm for PTS and proves its optimality. Section 6 considers the complexity of the discussed optimal priority assignment algorithms. Section 7 shows our empirical evaluation results. Finally, Section 8 concludes the paper.

2. Task Model

We use the same task model as the one used in the traditional preemption threshold scheduling [1, 3, 25, 26]. Specifically, a system has a fixed set of tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i$ has a fixed period $T_i$, a fixed relative deadline $D_i$, and a known worst-case execution time $C_i$. There is no restriction such that each task's deadline should be shorter than its period. We also adopt the "integer time model" of [9], where all timing parameters are assumed to be nonnegative integer values.

Each task $\tau_i$ also has a fixed priority $p_i$ and a preemption threshold $pt_i$ where $p_i$ is assigned by a specific priority assignment algorithm and $pt_i$ is assigned by OPT-ASSIGN-THRESHOLD() of [1]. Each task has a distinct priority value: each task has a different priority value. Each task set $\Gamma$ has $|\Gamma|$ distinct priority orderings for its tasks. Accordingly, the set of distinct priority orderings has cardinality $|\Gamma|!$, which we denote $PO^\Gamma = \{PO_1, PO_2, \ldots, PO_{|\Gamma|!}\}$. We denote the resultant priority ordering generated by a specific priority assignment algorithm ALGORITHM() as $PO_A$. With this, a specific priority ordering $PO_n$ is a sequence of priorities for tasks in $\Gamma$, which we denote as $PO_n = \langle p_1^n, p_2^n, \ldots, p_n^n \rangle$. The inverse mapping of each priority ordering $PO_n$ is a task ordering from the lowest priority to the highest priority, which we denote as $PO_n^{-1} = TO_n = \langle i, j, \ldots, k \rangle$ where each number represents a task index. We also denote the inverse mapping of task ordering $TO_n$ as $TO_n^{-1} = PO_n$.

We denote a higher priority with a larger value: 1 is the lowest priority value and $|\Gamma|$ is the highest priority value. Note that it is meaningful to assign a task a preemption threshold that is no less than its regular priority since a preemption threshold is used as an effective run-time priority to control unnecessary preemptions [1]: which means that $\forall \tau_i, pt_i \geq p_i$. Notation section summarizes the notations and associated descriptions used in this paper.

2.1. Feasibility Analysis. As the feasibility test under PTS, we adopt the worst-case response time analysis equations of [9]. The original equations were introduced by [1] and their errors were fixed by [27]. These results were refined by [26], whose results in turn were concisely arranged by [9]. We rewrite the relevant equations of [9] for calculating the worst-case response time $R_i$ of task $\tau_i$ as follows:

$$R_i = \max_{q \in \{1, \ldots, Q_i\}} \left\{ F_{i,q} - (q - 1) \cdot T_i \right\}, \quad (1)$$

$$Q_i = \left\lfloor \frac{L_i}{T_i} \right\rfloor, \quad (2)$$

$$L_i = B_i + \sum_{\forall j, j \neq i, p_j \geq p_i} \left( \left\lfloor \frac{L_j}{T_j} \right\rfloor \cdot C_j \right). \quad (3)$$
Table 1: A walk-through example task set. Bold and italic R's represent infeasible response times.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>( C_i )</th>
<th>( T_i )</th>
<th>( D_i )</th>
<th>( p_i )</th>
<th>( p_t )</th>
<th>( R_i )</th>
<th>( p_i )</th>
<th>( p_t )</th>
<th>( R_i )</th>
<th>( p_i )</th>
<th>( p_t )</th>
<th>( R_i )</th>
<th>( p_i )</th>
<th>( p_t )</th>
<th>( R_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>8 43</td>
<td>36</td>
<td>1</td>
<td>4 31</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>1</td>
<td>4</td>
<td>31</td>
<td>2</td>
<td>4</td>
<td>30</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>4 33</td>
<td>33</td>
<td>2</td>
<td>4 30</td>
<td>2</td>
<td>2</td>
<td>23</td>
<td>3</td>
<td>3</td>
<td>25</td>
<td>3</td>
<td>3</td>
<td>25</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>5 48</td>
<td>31</td>
<td>3</td>
<td>3 26</td>
<td>3</td>
<td>3</td>
<td>19</td>
<td>2</td>
<td>4</td>
<td>30</td>
<td>1</td>
<td>4</td>
<td>31</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>( \tau_4 )</td>
<td>7 14</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>1</td>
<td>5</td>
<td>24</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>4</td>
</tr>
</tbody>
</table>

(i) DMPO: deadline monotonic priority ordering + OPT-ASSIGN-THRESHOLD() employed in [9].
(ii) GREEDY-SA(): Greedy() + SimulatedAnnealing() proposed in [3].

\[ F_{i,j} = S_{i,j} + C_i \]
\[ + \sum_{j,j',p_r} \left( \left[ \frac{F_{i,j}}{T_j} \right] - \left(1 + \left[ \frac{S_{i,j}}{T_j} \right] \right) \right) \cdot C_{j'} \]  
(4)

\[ S_{i,j} = B_i + (q - 1) \cdot C_i + \sum_{j,j',p_r} \left(1 + \left[ \frac{S_{i,j}}{T_j} \right] \right) \cdot C_{j'} \]

B_i = max \{C_j - 1 | \forall j, pt_j \geq p_i > p_j\},

where \( L_i \) is the longest level-\( p_i \) busy period [28], \( q \) is the index of instances of task \( \tau_i \) within \( L_i \), \( Q_i \) is the last index of instances of task \( \tau_i \) within \( L_i \), \( F_{i,j} \) is the finish time of the \( q \)th instance of task \( \tau_i \), \( S_{i,j} \) is the start time of the \( q \)th instance of task \( \tau_i \), and \( B_i \) is the worst-case blocking time of task \( \tau_i \). Whenever a variable appears on both sides of an equation (i.e., \( L_i \) in (3) and \( F_{i,j} \) in (4)), its value can be found by iterating until the value converges [27]. Refer to [9] for the appropriate initial values for the iterations. With this, we define formally the feasibility of a task or a task set as follows.

**Definition 1** (task feasibility). Task \( \tau_i \) with the assignment of \( p_i \) and \( pt_i \) is feasible \( \iff R_i \leq D_i \).

**Definition 2** (task set feasibility). Task set \( \Gamma \) with priority ordering \( PO_{\Gamma} = (p_{i,1}^\Gamma, p_{i,2}^\Gamma, \ldots, p_{i,n}^\Gamma) \) or task ordering \( TO_{\Gamma} = PO_{\Gamma}^{-1} \) is feasible \( \iff \) every \( \tau_i \) in \( \Gamma \) is feasible such that \( \tau_i \) with \( p_{i,i}^\Gamma \) in \( PO_{\Gamma} \) and \( pt_i \) determined by OPT-ASSIGN-THRESHOLD() is feasible.

2.2. A Walk-Through Example Task Set. As a walk-through example, we use the task set in Table 1 that is composed of four tasks. The deadline monotonic priority ordering (DMPO) is optimal in the fully preemptive fixed priority scheduling [22] and is so even though there are blockings if there is no jitter [21]. Therefore, the approach of assigning priorities using DMPO and then assigning preemption thresholds using OPT-ASSIGN-THRESHOLD() of [1] is widely used in practice, which was also employed in [9] when comparing PTS with other limited preemptive scheduling policies. Since the task indexes of the example task set happen to be in the deadline monotonic decreasing order, the resultant priority ordering of DMPO is \( PO_{\Gamma_D} = (1, 2, 3, 4) \). However, as shown in Table 1, this priority ordering makes task \( \tau_4 \) miss its deadline since \( (R_4 = 14) > (D_4 = 11) \). Figure 1(a) demonstrates such a deadline miss: the fifth instance of task \( \tau_4 \) completes at time point 69 while its absolute deadline is \( (5 - 1) \cdot T_4 + D_4 = 67 \). Note that the worst-case response time cannot be obtained at the critical instant [29] when there is a nonpreemptiveness of tasks [9].

On the other hand, [3, 24] proposed heuristic priority assignment algorithms for PTS. Reference [3] proposed an algorithm that combines Greedy() and SimulatedAnnealing(), which we refer to GREEDY-SA() in this paper. Reference [24] proposed PA-DMMPT(), which means priority assignment algorithm assuming Deadline Monotonic and Maximum Preemption Threshold for the remaining tasks in the unassigned task set. The resultant priority ordering of GREEDY-SA() and PA-DMMPT() is, respectively, \( PO_{\Gamma_G} = (4, 2, 3, 1) \) and \( PO_{\Gamma_D} = (1, 3, 2, 4) \) as shown in Table 1. These priority orderings also make task \( \tau_4 \) miss its deadline as shown in Table 1.

Figures 1(b) and 1(c) demonstrate such deadline misses of task \( \tau_4 \). In Figure 1(b), the first, the second, and the fifth instances of task \( \tau_4 \) complete at time points 24, 31, and 69, respectively, while their absolute deadlines are \( D_4 = 11 \), \( (2 - 1) \cdot T_4 + D_4 = 25 \), and \( (5 - 1) \cdot T_4 + D_4 = 67 \), respectively. In Figure 1(c), the fifth instance of task \( \tau_4 \) completes at time point 69 while its absolute deadline is \( (5 - 1) \cdot T_4 + D_4 = 67 \). As such, widely used DMPO or heuristic priority assignment algorithms may not produce a feasible priority assignment while there is actually one, as will be shown in Section 3.1. This motivated our work.

3. Previously Proposed Optimal Priority Assignment Algorithms for PTS

This section analyzes previously proposed optimal priority assignment algorithms for PTS: TRAVERSE() [13] and SEARCH() [1].

3.1. TRAVERSE() Algorithm. Algorithm I(a) shows the pseudo code for TRAVERSE(). As shown in Algorithm I(a), TRAVERSE() depends on its recursive subroutine, TRAVERSE(), which has three parameters: (1) \( prio \), the next priority to assign, (2) \( UnAssigned \), the set of tasks that are waiting for priority assignment, and (3) \( \Gamma \), the set of total tasks (line (3)). TRAVERSE() assigns preemption thresholds by calling OPT-ASSIGN-THRESHOLD() of [1] when a complete priority ordering is generated (line (4)). When any preemption threshold assignment is not feasible, OPT-ASSIGN-THRESHOLD() returns fail. With
Figure 1: Schedule produced for the walk-through example task set of Table 1 by various priority assignment algorithms. Note that some instances of task $\tau_4$ in (a)–(d) miss their deadlines while every task in (e) does not miss its deadline.
The remaining part is composed of a loop that recursively invokes _TRAVERSE(): each task in \textit{UnAssigned} at line (5) is assigned the priority \textit{prio} at line (6) and remaining unassigned tasks (\textit{UnAssigned} - \{\tau\}) are recursively assigned \textit{prio} + 1 at line (7). Note that the recursive invocation of _TRAVERSE() returns only when the return value of the recursive invocation is \textit{success} (line (7)). We call this priority assignment the tentative priority assignment since if the return value of the recursion is \textit{fail}, the next task in \textit{UnAssigned} is tried for assigning priority \textit{prio}.

With such tentative priority assignments, _TRAVERSE() recursively generates a priority assignment tree for the set of tasks with task orderings from the lowest priority to the highest priority, which is similar to the priority permutation tree of [21]. Figure 2(a) shows the priority assignment tree of _TRAVERSE() for the walk-through example task set of Table 1. In Figure 2, each solid-lined circle node represents a tentative priority assignment to a task and its depth corresponds to its assigned priority. Each triangle node represents an invocation of OPT-ASSIGN-THRESHOLD(): white one for infeasible (failed) assignment return and black one for feasible (successful) assignment return. Each path from the root to a leaf node corresponds to a possible priority ordering. Task ordering \langle 3, 2, 1, 4 \rangle has a dangled black triangle node, which indicates a feasible preemption threshold assignment. Task ordering TO\_\tau \_\gamma = \langle 2, 3, 1, 4 \rangle has a dangled black triangle node, which indicates a feasible preemption threshold assignment. Task ordering TO\_\tau \_\gamma corresponds to the priority ordering PO\_\tau \_\gamma = \langle 2, 3, 1, 4 \rangle for _TRAVERSE() of Table 1. As shown in Table 1, _TRAVERSE() makes no tasks miss their deadlines, which is also demonstrated in Figure 1(e). While [13] did not formally prove the optimality of _TRAVERSE(), we can easily see that _TRAVERSE() is optimal since it traverses all possible priority orderings in PO\_\gamma until it finds a feasible priority ordering.

**Algorithm 1:** Pseudo code for (a) _TRAVERSE() [13] and (b) _SEARCH() [1].

```
(1) TRAVERSE (Γ: set of tasks)
(2) return _TRAVERSE(1, Γ, Γ);

(3) _TRAVERSE (prio: priority, UnAssigned: set of tasks, Γ: set of tasks)
(4) if (UnAssigned = {}) return OPT-ASSIGN-THRESHOLD(Γ);
(5) foreach (τ ∈ UnAssigned) {
     (6) pᵢ ← prio;
     (7) if (_TRAVERSE(prio + 1, UnAssigned - {τ}, Γ) = success) return success;

     (8) // end-foreach
     (9) return fail;

(a) TRAVERSE() [13]

(1) SEARCH (Γ: set of tasks)
(2) return _SEARCH(1, Γ, Γ);

(3) _SEARCH (prio: priority, UnAssigned: set of tasks, Γ: set of tasks)
(4) if (UnAssigned = {}) return OPT-ASSIGN-THRESHOLD(Γ);
(5) foreach (τ ∈ UnAssigned) {
     // WCRT(τ, prio): Rᵢ with (pᵢ ← prio) under fully preemptive fixed priority scheduling
     // assuming all tasks in UnAssigned have the highest priority.
     (6) Delayᵢ ← WCRT(τ, prio) - Dᵢ;
     (7) if (Delayᵢ ≤ 0) {
         (8) pᵢ ← prio;
         (9) return _SEARCH(prio + 1, UnAssigned - {τ}, Γ);
     }

     (10) // end-if
     (11) // end-foreach
     (12) SortedList ← ascendingSort(UnAssigned, Delayᵢ);
     (13) RefinedList ← Refine(SortedList); // eliminating infeasible tasks even with the highest preemption threshold
     (14) foreach (τ ∈ RefinedList) {
         (15) pᵢ ← prio;
         (16) if (_SEARCH(prio + 1, UnAssigned - {τ}, Γ) = success) return success;
     }

     (17) // end-foreach
     (18) return fail;

(b) SEARCH() [1]
```

this, returning at the last lines (line (9)) happens when a specific priority ordering is not feasible.
Besides, it is a simple application of traverse_orderings() [21], which is for fully preemptive fixed priority scheduling and its optimality was proved.

However, TRAVERSE() is very inefficient since it generates $|Γ|!$ tentative priority orderings in the worst-case, each of which requires maximally $|Γ|^2$ feasibility tests [3] via OPT-ASSIGN-THRESHOLD(). For example, the priority assignment tree of Algorithm 3(a) requires 91 feasibility tests as the numbers besides triangle nodes indicate. Such a large number of feasibility tests can be significantly reduced in our proposed algorithm in Section 5.

3.2. SEARCH() Algorithm. Algorithm 1(b) shows the pseudo code for SEARCH(), which becomes exactly the same as
TRAVERSE() if we remove lines (5)∼(13) and replacing RefineList with UnAssigned at line (14). The additional part of SEARCH() is composed of one loop (lines (5)∼(II)) and refining UnAssigned to prepare RefineList (lines (12) and (13)). In contrast to the priority assignment loop of TRAVERSE(), this additional first loop in SEARCH() assigns priority prio to task \( \tau_i \) only when Delay\( _i \) (calculated at line (6)) is not larger than zero (line (7)) and unconditionally returns from the recursion (line (9)). We call this priority assignment in the first loop as the **assertive priority assignment** since once priority prio is assigned to \( \tau_i \), the other unassigned tasks are not tested for assigning priority prio.

The second loop of SEARCH() (lines (14)∼(18)) performs the tentative priority assignment as TRAVERSE() but it works for RefineList instead of UnAssigned. The RefineList is generated by sorting tasks in UnAssigned in the ascending order of Delay\( _i \) (line (12)) and eliminating infeasible tasks even with the highest preemption threshold assignment (line (13)).

With the assertive and tentative priority assignments, SEARCH() generates a priority assignment tree like Figure 2(b), which is for the walk-through example task set of Table 1. In the tree, each solid-lined square node represents an assertive priority assignment to a task while each dashed-lined node represents pruning a priority assignment to a task. In Figure 2(b), two complete task orderings were generated: \( \{2,1,3,4\} \) and \( \{3,1,2,4\} \). Both orderings failed to assign feasible preemption thresholds to tasks as the dangled white triangle nodes indicate. The last task ordering TO\( _S = \{3,1,2,4\} \) corresponds to the priority ordering PO\( _S = \{2,3,1,4\} \) for SEARCH() of Table 1. As shown in Table 1, SEARCH() makes task \( \tau_4 \) miss its deadline since \( R_4 = 14 \) > \( D_4 = 11 \). Figure 1(d) demonstrates such a deadline miss of task \( \tau_4 \) where the second instance of task \( \tau_4 \) completes at time point 26 while its absolute deadline is \( T_4 + D_4 = 25 \). However, the walk-through example task set has a feasible priority ordering PO\( _T = \{3,2,1,4\} \) from TRAVERSE() algorithm as we have shown in Section 3.1. This clearly shows SEARCH() is not an optimal priority assignment algorithm for PTS.

If we compare the priority assignment trees of TRAVERSE() and SEARCH() in Figures 2(a) and 2(b), we can see that TO\( _S = \{3,1,2,4\} \) is the 13th task ordering generated by TRAVERSE(). The reason why SEARCH() failed is due to the assertive priority assignment of \( p_{t_1}^1 \leftarrow 2 \) with the 21st feasibility test, which we marked with a red dashed-line circle in Figure 2(b). We correct SEARCH() in the next section.

### 4. Corrected SEARCH() Algorithm

In this section, we propose CORRECTED-SEARCH() algorithm that corrects SEARCH() algorithm. We first develop a theorem that is required in correcting SEARCH().

**Lemma 3.** Under PTS, if priority \( p_t \) of task \( \tau_i \) is fixed, its worst-case response time \( R_i \) does not decrease when its preemption threshold \( p_t \) is lowered.

**Proof.** While \( R_i \) is calculated from (1)∼(6), \( p_t \) is only included in (4) for calculating \( F_{i,t} \), specifically in the last term of the right side of (4): \( \sum_{i, p > p_t} [(F_{i,t}/T_{i,t}) - (1 + [S_{i,t}/T_{i,t}])] \cdot C_i \). We refer to this term as \( I \) for the convenience of proving. \( I \) is the summation of the interference time by task \( \tau_i \) that it works for RefineList instead of UnAssigned.

From Theorem 4, pruning at line (5) makes RefineList instead of UnAssigned. Theorem 4.

Theorem 4. Under PTS, if task \( \tau_i \) with priority \( p_t^i \) in priority ordering \( \Gamma \) is infeasible, task set \( \Gamma \) with \( p_t^i \) is also infeasible.

**Proof.** From Definition 1, task \( \tau_i \) is feasible if and only if \( R_i \leq D_i \). From Lemma 3, if \( p_t \) is fixed (as \( p_t^i \)), \( p_t = |\Gamma| \) guarantees the minimal \( R_i \) for \( \tau_i \). Accordingly, if task \( \tau_i \) with fixed \( p_t \) and \( p_t = |\Gamma| \) is infeasible, no other preemption threshold assignment can make \( R_i \) reduced, that is, \( \tau_i \) feasible. From Definition 2, \( \Gamma \) with \( p_t^i \) is feasible if and only if all tasks with \( p_t^i \) in \( \Gamma \) are feasible. This proves the theorem.

Algorithm 2(a) shows the pseudo code of CORRECTED-SEARCH(), which corrects SEARCH() in Algorithm 1(b) as follows:

1. (i) deletion of the first loop of the assertive priority assignment and sorting tasks (lines (5)∼(12) of Algorithm 1(b)),
2. (ii) replacement of SortedList with UnAssigned (line (13) of Algorithm 1(b)),
3. (iii) adding \( p_t \leftarrow \text{prior} \) after \( p_t \leftarrow \text{prior} \) (line (15) of Algorithm 1(b)),
4. (iv) replacement of OPT-ASSIGN-THRESHOLD(\( \Gamma \)) at line (4) with RESTORING-OPT-ASSIGN-THRESHOLD(\( \Gamma \)) of Algorithm 2(b).

First, CORRECTED-SEARCH() does not employ the first loop of SEARCH(), the assertive priority assignment, which is the main cause of the nonoptimality of SEARCH() as shown in the previous section. If we change the assertive priority assignment to a tentative priority assignment by replacing line (9) with line (16) in Algorithm 1(b), it just induces additional feasibility tests (due to invocation of operation WCRT()) when a given task set is indeed infeasible. In other words, the first loop just becomes a performance bottleneck even though it is corrected and thus we remove it.

Second, without the first loop of SEARCH(), Delay\( _i \) is not calculated and thus the task sorting of line (12) of Algorithm 1(b) is not also employed. Note that the second loop of SEARCH() is exactly the same as TRAVERSE() except that it works for RefineList instead of UnAssigned. From Theorem 4, pruning at line (5) that makes RefineList
is valid. With this, we can easily see that CORRECTED-SEARCH() is optimal since TRAVERSE() is optimal.

Third, we assign task $\tau_i$ preemption threshold $pt_i$ as the same value of its priority at line (8) whenever its priority $p_i$ is assigned (line (7)). This is to make sure that any priority assigned task is preemptable by higher priority tasks. For the feasibility test for refining $UnAssigned$ by subroutine Refine() at line (5) to work correctly, it should be guaranteed
that any task in UnAssigned can preempt the preassigned lower priority tasks in Γ − UnAssigned. SEARCH() does not need this preemption threshold assignment due to WCRT() invocation at line (6) of Algorithm 1(b) where every task’s preemption threshold is set as its priority.

Finally, RESTORING-OPT-ASSIGN-THRESHOLD() in Algorithm 2(b) restores the preemption threshold of each task to its priority in the case of the failed (infeasible) preemption threshold assignment return. Without this correction, after OPT-ASSIGN-THRESHOLD() has assigned preemption thresholds to tasks, preemption threshold values of tasks are contaminated and thus subroutine Refine() at line (5) cannot work correctly.

Figure 2(c) shows the priority assignment tree of CORRECTED-SEARCH() for the walk-through example task set of Table 1. In the figure, six complete task orderings were generated. The last task ordering TOC = (3, 2, 1, 4) is the same as TOR, and the resultant feasible task ordering of TRAVERSE(). CORRECTED-SEARCH() requires 74 feasibility tests while TRAVERSE() requires 91 feasibility tests as the numbers beside triangle nodes indicate in Figures 2(a) and 2(c). As such, it is obvious that the overall performance of CORRECTED-SEARCH() is much better than that of TRAVERSE() since CORRECTED-SEARCH() prunes infeasible paths while TRAVERSE() does not. We develop the more performance enhanced algorithm in the next section.

5. PRUNED-TRAVERSE() Algorithm

SEARCH() is not optimal since it prunes even feasible priority orderings. As such, pruning exhaustively without harming the optimality is important. In this section, we propose our optimal priority assignment algorithm for PTS, which we named PRUNED-TRAVERSE(). We first develop required theorems for our proposed algorithm.

Lemma 5. Under any priority assignment algorithm that assigns distinctive priorities to tasks for PTS, if the priority pₖ of a task τₖ is lowered to pₖ’ (pₖ’ < pₖ), there exists at least one lower priority task τⱼ (pⱼ < pₖ’) that heightens its priority pⱼ to pⱼ’ with pⱼ’ > pⱼ.

Proof. The lemma can be easily proved by observing task orderings in priority assignment trees like Figure 2(a). If task τ₁ is moved to a lower priority level (the higher place), at least one of the lower priority tasks should be moved to a higher priority level (the lower place) since the total priority levels are fixed.

Theorem 6. Under PTS, if preemption threshold pt of task τᵢ is fixed, its worst-case response time Rᵢ does not decrease when its priority pᵢ is lowered.

Proof. For the convenience of proving, we refer to the last terms of the right sides of (4) and (5) as Iᵢ^F and Iᵢ^S, respectively. We prove the theorem by contradiction. Suppose that Rᵢ of task τᵢ decreases to Rᵢ’ when its priority pᵢ is lowered to pᵢ’; Rᵢ’ < Rᵢ and pᵢ’ < pᵢ. For Rᵢ to decrease, Sᵢ<j should decrease from (1). For Sᵢ<j to decrease, Sᵢ,j or Iᵢ<j^S should decrease from (4). Since the condition for calculating Iᵢ^F is pⱼ > pt, the change of priority pⱼ does not affect Iᵢ<j^F. Therefore, Sᵢ,j should decrease to Sᵢ,j’ such that Sᵢ,j − Sᵢ,j’ > 0. Then, from (5), (Iᵢ<j^S − Iᵢ<j^F) > 0 follows where Bᵢ’ and Iᵢ<j^S’ are changed values of Bᵢ and Iᵢ<j^S due to the lower priority. Then, (Bᵢ − Bᵢ’) > 0 follows.

From Lemma 5, if pᵢ is lowered to pᵢ’, there exists at least one lower priority task τⱼ (pⱼ < pᵢ) that heightens its priority pⱼ to pⱼ’ such that pⱼ’ > pⱼ. Let the set of such additionally introduced higher priority tasks after the priority of task τᵢ is lowered be AddedHPⱼ = {τⱼ | pⱼ < pᵢ and pⱼ’ > pⱼ}. The maximum possible value of (Bᵢ − Bᵢ’) is achieved when Bᵢ’ = 0, which infers that at least one task in AddedHPⱼ contributes in making Bᵢ. Let such a task in AddedHPⱼ be task τⱼ; that is Bⱼ = Cⱼ. Then, max(Bᵢ − Bᵢ’) = Cⱼ follows. On the other hand, from (5), it follows that Iᵢ<j^S = Iᵢ<j^F = \sum_{j,p∈AddedHP,j \neq i} [Sⱼ/p(Tⱼ) \cdot Cⱼ ≤ Cⱼ. Then, (Bᵢ − Bᵢ’) ≤ (Iᵢ<j^S − Iᵢ<j^F) follows.

This is a contradiction, which proves the theorem.

Theorem 7. Under PTS, if task τᵢ with priority pᵢᵢ in priority ordering POᵢ and the highest preemption threshold ptᵢ = |Γ| is infeasible, task set Γ with another priority ordering POᵢ that assigns task τᵢ the lower priority pᵢᵢ’ (such that pᵢᵢ’ < pᵢᵢ) is also infeasible.

Proof. Let the worst-case response time of task τᵢ with ptᵢ = |Γ| and pᵢᵢ’ be Rᵢ’ and Rᵢ. Let the worst-case response time of task τᵢ with ptᵢ = |Γ| and pᵢᵢ’ be Rᵢ’. From Lemma 3, both Rᵢ and Rᵢ’ are the minimal possible worst-case response time with each given priority. From Theorem 6, Rᵢ’ ≥ Rᵢ follows. Since task τᵢ with pᵢᵢ’ and ptᵢ = |Γ| is infeasible, Rᵢ’ > Dᵢ follows from Definition 1. Therefore, Rᵢ’ > Dᵢ follows. Accordingly, task τᵢ with pᵢᵢ’ and ptᵢ = |Γ| is infeasible. Consequently, from Theorem 4, task set Γ with POᵢ is also infeasible.

Now we propose PRUNED-TRAVERSE() that extends TRAVERSE() by exhaustively pruning infeasible paths. PRUNED-TRAVERSE() prunes such priority ordering POᵢ that assigns task τᵢ priority pᵢᵢ when the following condition is true:

C₁ ∨ C₂, (7)

where C₁ is ptᵢ = |Γ| ∧ Rᵢ > Dᵢ, and C₂ is pᵢᵢ < max{pⱼ | C₁}. Apparently, Condition C₁ is from Theorem 4 and Condition C₂ is from Theorem 7.

Algorithm 3 shows the pseudo code for PRUNED-TRAVERSE(). Like TRAVERSE(), TRAVERSE() depends on its subroutine,, PRUNED-TRAVERSE(), which has the same parameters as _TRAVERSE(): _pri, UnAssigned, and Γ (line (9)). Like _TRAVERSE(), _PRUNED-TRAVERSE() assigns priorities to tasks from the lowest priority 1 to the highest priority |Γ|; it assigns each task τᵢ in UnAssigned (line (10)) priority _pri (line (12)) and the
remaining unassigned tasks \((\text{UnAssigned} - \{\tau_i\})\) are recursively assigned priority \(\text{prio} + 1\) at line (21).

Unlike \(\text{TRAVERSE}\), \(\text{PRUNED-TRAVERSE}\) first sorts tasks in a deadline monotonic decreasing order since DMPO also works well in many cases (line (2)). \(\text{PRUNED-TRAVERSE}\) introduces for task \(\tau\) new attribute \(\text{infeasiblePrioMax}_\tau = \max\{p, | C1\}\), which is the right side of the Condition \(C2\). It initializes \(\text{infeasiblePrioMax}_\tau\) as zero at line (4).

Unlike \(\text{TRAVERSE}\), \(\text{PRUNED-TRAVERSE}\) needs to perform a feasibility test for task \(\tau_i\) (getting \(R_i\)) while assigning each priority in order to prune infeasible paths. For this, we need to make sure that all tasks in \(\text{UnAssigned}\) have higher priorities than the priorities of the previously priority assigned tasks. For this, \(\text{PRUNED-TRAVERSE}\) assigns the highest priority \(|\Gamma|\) to all unassigned tasks initially at line (5). \(\text{PRUNED-TRAVERSE}\) also does so conditionally at line (26) whenever the next priority \((\text{prio} + 1)\) assignment fails at line (25). Preemption thresholds of all tasks are also initialized as the same value of their priorities at line (6).

Before assigning task \(\tau_i\) priority \(\text{prio}\) (line (12)), \(\text{PRUNED-TRAVERSE}\) first prunes any infeasible priority ordering at line (11) if \(\text{prio}\) is smaller than \(\text{infeasiblePrioMax}_\tau\), which is Condition \(C2\). Condition \(C1\) is applied at lines (13) and (19) and the priority ordering with Condition \(C1\) is pruned at line (22). Line (20) is for restoring the priority of the pruned task. Line (21) is for updating \(\text{infeasiblePrioMax}_\tau\). If task \(\tau_i\) is assigned priority \(\text{prio}\) without being pruned, its preemption threshold is assigned the same as its priority at line (24), without which PTS becomes the fully nonpreemptive scheduling (NPS) due to the highest preemption threshold assignment at line (13).

Once \(\text{prio}\) of the maximum priority \(|\Gamma|\) is assigned (line (14)), which means that a complete priority ordering is generated, \(\text{PRUNED-TRAVERSE}\) assigns preemption thresholds by invoking \(\text{RESTORING-OPT-ASSIGN-THRESHOLD}\) in Algorithm 2(b). Note that there are two differences with \(\text{TRAVERSE}\) in assigning preemption thresholds: (1) invoking \(\text{RESTORING-OPT-ASSIGN-THRESHOLD}\) instead of \(\text{OPT-ASSIGN-THRESHOLD}\) and (2) assigning preemption thresholds to tasks once the maximum priority is assigned instead once \(\text{UnAssigned}\) is \(\emptyset\). (1) is for the proper pruning operation as we explained in proposing \(\text{CORRECTED-SEARCH}\) in Section 4. (2) is for reducing one recursive function call for a performance benefit: one recursive function call is reduced for each complete priority ordering in \(\text{PRUNED-TRAVERSE}\) compared to \(\text{TRAVERSE}\) or \(\text{CORRECTED-SEARCH}\).

When any preemption threshold assignment is not feasible, \(\text{RESTORING-OPT-ASSIGN-THRESHOLD}\) returns fail. In that case, \(\text{infeasiblePrioMax}_\tau\) is set to \(|\Gamma|\) at line (16). This is because the fail return of \(\text{RESTORING-OPT-ASSIGN-THRESHOLD}\) infers that the highest priority assigned task \(\tau_i\) is not feasible due to some blocking task whose preemption threshold should be raised for it to be feasible. The recursive invocation of \(\text{PRUNED-TRAVERSE}\) returns only when the return value of the recursive invocation is \(\text{success}\) (line (25)). With this, returning at the last lines (line (28)) happens when a specific ordering is not feasible.

\(\text{PRUNED-TRAVERSE}\) requires careful restorations of priorities and preemption thresholds of tasks, which are repetitively and tentatively assigned. Since \(\text{SEARCH}\) and \(\text{CORRECTED-SEARCH}\) prune infeasible paths within subroutine \(\text{Refine}\), such restoration is easier and less prone. However, \(\text{PRUNED-TRAVERSE}\) prunes infeasible paths as earlier as possible for the better efficiency and thus requires careful restoring operations. Priorities are restored before pruning at line (20) and after the failed priority assignment at line (26). Preemption thresholds are restored in \(\text{RESTORING-OPT-ASSIGN-THRESHOLD}\) at line (15). Assigning the preemption threshold as same as the priority of the priority assigned task at line (24) is also important for the proper pruning.

Now we prove the optimality of \(\text{PRUNED-TRAVERSE}\).

**Theorem 8.** \(\text{PRUNED-TRAVERSE}\) is optimal for PTS: it finds a feasible scheduling attributes assignment if there exists one.

**Proof.** \(\text{PRUNED-TRAVERSE}\) extends \(\text{TRAVERSE}\) by pruning infeasible priority orderings with Condition \(C1\) or \(C2\). Condition \(C1\) is valid from Theorem 4 and condition \(C2\) is valid from Theorem 7. Consequently, since \(\text{TRAVERSE}\) is optimal, \(\text{PRUNED-TRAVERSE}\) is optimal.

Figure 2(d) shows the priority assignment tree of \(\text{PRUNED-TRAVERSE}\) for the walk-through example task set of Table 1. In the figure, a gray dashed-lined circle represents pruning a priority ordering without any feasibility test, which is achieved by application of Condition \(C2\) from Theorem 7. By comparing Figures 2(c) and 2(d), we can easily see this additional pruning helps much in reducing the number of feasibility tests: the feasibility tests of (7), (21)–(39), (55)–(62) in Figure 2(c) do not happen in Figure 2(d). We can also see that the earlier pruning with condition \(C1\) of \(\text{PRUNED-TRAVERSE}\) instead of using subroutine \(\text{Refine}\) of \(\text{CORRECTED-SEARCH}\) reduces the number of feasibility tests: the feasibility tests of (4), (42), (64) in Figure 2(c) do not happen in Figure 2(d).

The resultant last task ordering \(\text{T}_{\tau\tau} = \{3, 2, 1, 4\}\) is the same as \(\text{T}_{\tau\tau}\) and \(\text{T}_{\tau\tau}\), which are the resultant feasible task ordering of \(\text{CORRECTED-SEARCH}\) and \(\text{TRAVERSE}\), respectively. \(\text{PRUNED-TRAVERSE}\) produces four complete priority orderings and requires 47 feasibility tests while \(\text{CORRECTED-SEARCH}\) produces six complete priority orderings and requires 74 feasibility tests. As such, it is obvious that the overall performance of \(\text{PRUNED-TRAVERSE}\) is much better than that of \(\text{CORRECTED-SEARCH}\) since \(\text{PRUNED-TRAVERSE}\) prunes more infeasible paths than \(\text{CORRECTED-SEARCH}\) exploiting Theorem 7 and the earlier pruning. We show the empirical performance comparison results in Section 7.

### 6. Complexity

Let \(n = |\Gamma|\) and \(E\) the nonpolynomial [20] complexity of the feasibility test (calculating \(R_i\) from (1)–(6)). Since the pruning operation is conditional, all these algorithms
in the worst-case produce \( n! \) priority orderings of \( \text{PO}^\Gamma \), each of which requires \( O(n^r) \) feasibility tests [3] via OPT-ASSIGN-THRESHOLD(). With this, TRAVERSE() has the complexity of \( O(E \cdot n! \cdot n^r) \).

On the other hand, each pruning operation of CORRECTED-SEARCH() and PRUNED-TRAVERSE() requires one feasibility test for each tentative priority assignment or pruning (circle) node. The number of circle nodes in a priority assignment tree in the worst-case is \( n + n \cdot (n - 1) + n \cdot (n - 1) \cdot (n - 2) + \cdots + n! \), which is \( O(n!) \). Therefore, the worst-case complexity of CORRECTED-SEARCH() and PRUNED-TRAVERSE() is \( O(E \cdot n! \cdot n^r) + O(E \cdot n!) = O(E \cdot n! \cdot n^r) \), which is the same as that of TRAVERSE(). However, the pruning operation obviously works as a branch and bound mechanism and derives much better performances in most cases as the following empirical comparison shows.

### 7. Empirical Performance Evaluations

We set the performance metrics of priority assignment algorithms for PTS as (1) schedulability as the ratio of feasible task sets and (2) actual runtimes for executing algorithms. The experiments for getting the actual runtimes of the algorithms were done on Intel Core i7-4770, 3.40 GHz with 8 GB RAM. We set the total utilization of each task set as \( U = 0.9 \). This utilization represents high-demanding workloads, which makes the feasibility of a given task set greatly dependent on a proper priority assignment algorithm. Note that the utilization bound \( U_B \) under the rate monotonic scheduling (RMS) [29] when \( |\Gamma| = n \) is \( U_B = n \cdot (\sqrt{2} - 1) \). For example, if \( |\Gamma| = 10 \) and \( U \leq 0.72 \) (\( \approx 10 \cdot (\sqrt{2} - 1) = U_B \)), we even do not need PTS if task deadlines are the same as periods since the fully preemptive fixed priority scheduling with DMPO always makes the task set feasible.

We generated each task set in the same manner as [9]. Specifically, for a given total utilization \( U = 0.9 \), we generated each task's utilization \( u_i \) using UUniFast [30] algorithm. For each task \( t_i \), we generated \( C_i \) as a random integer uniformly distributed in the interval \([100, 500]\), \( T_i = C_i/u_i \), and \( D_i \) as a random integer uniformly distributed in the interval \([C_i + 0.5 \cdot (T_i - C_i), T_i] \). For each experiment with a specific parameter setting, we generated 2,000 task sets. To focus on the effectiveness of pruning operations of PRUNED-TRAVERSE(), each task set was ordered in the deadline monotonic decreasing order. Since TRAVERSE() took too much time when the number of tasks is large, we applied timeout for executing TRAVERSE() when \( |\Gamma| \geq 10 \). The actual runtimes of TRAVERSE() with timeout were set as the real actual runtime of TRAVERSE() when a task set was not feasible.

#### 7.1. Schedulability

Figure 3 shows the schedulability results with \( U = 0.9 \) and \( |\Gamma| = 10 \). Figure 3(a) shows the percentage ratio of feasible task sets and Figure 3(b) shows the Venn diagram for the number of feasible task sets. The ratio of feasible task sets was calculated as the number of feasible task sets divided by the total number of generated task sets. The resultant feasible task set ratios of TRAVERSE(), CORRECTED-SEARCH(), and PRUNED-TRAVERSE() were exactly the same and thus we omitted the results of TRAVERSE() and CORRECTED-SEARCH() in Figure 3.

Figure 3(a) clearly shows that the schedulability performance results of PA-DMMPT() [24], SEARCH() [1], GREEDY-SA() [3], DMPO [9], and PRUNED-TRAVERSE() are uniformly improved in order. Specifically, PA-DMMPT(), SEARCH(), GREEDY-SA(), DMPO, and PRUNED-TRAVERSE() could, respectively, schedule 50.6\%, 53.45\%, 59.9\%, 64.65\%, and 67.65\%. Note that PRUNED-TRAVERSE() outperforms than any other priority assignment algorithms, which clearly shows that the other algorithms are nonoptimal.

Figure 3(b) shows the Venn diagram for the number of feasible task sets in 2,000 task sets. As shown, PRUNED-TRAVERSE() could schedule 27 task sets that could not be scheduled by any other existing heuristic priority assignment algorithms. It is notable that each heuristic algorithm could schedule some task sets that could not be scheduled by the other heuristic algorithms. For example, SEARCH() could schedule two task sets that could not be scheduled by the other heuristic algorithms.

DMPO is a very efficient algorithm that requires almost no implementation efforts as well as computation time burden. Accordingly, it is practical to combine DMPO with another heuristic algorithm. Figure 3(b) shows that SEARCH(), GREEDY-SA(), and PA-DMMPT(), respectively, could schedule 5 (2 + 0 + 2 + 1), 19 (9 + 8 + 2 + 0), and 22 (11 + 1 + 2 + 8) task sets that could not be schedule by DMPO. On the other hand, DMPO + SEARCH(), DMPO + GREEDY-SA(), and DMPO + PA-DMMPT(), respectively, could schedule 64.9\%, 65.6\%, and 65.75\%. With this, we conclude that PA-DMMPT() is the most effective heuristic priority assignment algorithm as the best candidate to be combined with DMPO. Therefore, we compare its actual runtimes with our proposed algorithms in the next subsection. Note that any DMPO combined heuristic algorithm cannot become optimal due to the existence of task sets that can be scheduled only by PRUNED-TRAVERSE() as shown in Figure 3(b).

#### 7.2. Actual Runtimes

We compare the actual runtimes for executing optimal priority assignment algorithms and PA-DMMPT(), which is the most effective heuristic algorithm to be combined with DMPO as shown in the previous section. Figures 4(a) and 4(b) show the actual runtime results in seconds as box plots when \( U = 0.9 \) for \( |\Gamma| = 5 \) and \( |\Gamma| = 10 \), respectively. \( Y \) axis is in a log scale to better show the distributions of result values. Each box in a box plot shows data results between 25\% and 75\% of performance distribution. The middle line within each box shows the median value of the results while the filled circle mark shows the average value of the results.

Figures 4(a) and 4(b) clearly show that the actual runtime performance distribution results of TRAVERSE(), CORRECTED-SEARCH(), and PRUNED-TRAVERSE() are
uniformly improved in order. In Figure 4(a) with $|\Gamma| = 5$, the maximum and average runtimes were decreased by 72.6% and 87.0%, respectively, in CORRECTED-SEARCH() compared to the TRAVERSE() (from 0.84 sec to 0.23 sec and from 0.23 sec to 0.03 sec). PRUNED-TRAVERSE() further decreased the maximum and average actual runtime values of CORRECTED-SEARCH() by 8.7% and 33.3%, respectively (to 0.21 sec and to 0.02 sec).

Such performance differences become drastically large as the number of tasks $|\Gamma|$ increases. In Figure 4(b) with $|\Gamma| = 10$, the maximum and average actual runtimes of CORRECTED-SEARCH() compared to TRAVERSE() were decreased by 98.0% and 99.95%, respectively (from 67809 sec to 1331 sec and from 21947 sec to 10.73 sec). Moreover, PRUNED-TRAVERSE() decreased the maximum and average values of CORRECTED-SEARCH() by 61.2% and 58.3%, respectively (to 517 sec and to 4.47 sec).

On the other hand, we can see the median values of actual runtimes of TRAVERSE() were always smaller than any other algorithms. This is because TRAVERSE() does not perform

Figure 3: Schedulability with $U = 0.9$ and $|\Gamma| = 10$.

Figure 4: Actual runtimes in seconds when $U = 0.9$. 
any feasibility test for priority assignments that are required by the other algorithms. When a task set is feasible with DMPO, the actual runtimes of TRAVERSE() are the same as DMPO.

Figures 4(a) and 4(b) also show that the actual runtimes of PA-DMMPT() are much larger than PRUNED-TRAVERSE() and even larger than CORRECTED-SEARCH(). Specifically in Figure 4(b) with |Γ| = 10, the maximum and average actual runtimes of PA-DMMPT() compared to PRUNED-TRAVERSE() were increased by 1167% and 150% (from 517 sec to 6551 sec and from 4.47 sec to 11.18 sec). This is because PA-DMMPT() calculates task blocking limits by generating candidate pairs of start and finish times of tasks. As well investigated in [31], the execution requirements of such an approach that generates scheduling points grow exponentially with an increasing range of task periods. Our experimental task sets were generated in the same manner as [9], where the range of task periods increases as the number of tasks increases.

However, PRUNED-TRAVERSE() still cannot be used as an online feasibility test algorithm since its performance is also degraded much as the number of tasks increases. For example, for a task set Γ with U = 0.9 and |Γ| = 20, PRUNED-TRAVERSE() took 7 hours to determine that the task set was after all infeasible. Nevertheless, the above experimental results clearly show that PRUNED-TRAVERSE() outperforms any other optimal priority assignment algorithms as well as the best effective heuristic priority assignment algorithm for PTS.

8. Conclusion

Preemption threshold scheduling (PTS) has been widely accepted in the industrial domain for its effectiveness of scalable real-time embedded system design with the increased real-time schedulability (feasibility). However, without an available optimal scheduling attributes assignment algorithm (optimal in the sense that it is guaranteed to find a feasible scheduling attributes assignment if one exists), we cannot achieve the full benefits of PTS.

Since there exists an optimal and efficient O(n^3) preemption threshold assignment algorithm [1] that operates with fully assigned priority orderings, we need an optimal priority assignment algorithm for PTS. In this paper, we analyzed previously proposed optimal priority assignment algorithms for PTS: TRAVERSE() [13] and SEARCH() [1]. Using priority assignment trees, we showed the inefficiency of TRAVERSE() due to its lack of any pruning operation. We also showed the nonoptimality of SEARCH() due to its pruning of even feasible priority orderings.

We developed some theorems for safely and exhaustively pruning infeasible priority ordering paths while assigning priorities to tasks before assigning feasible preemption thresholds for PTS. Using these theorems, we corrected SEARCH() and presented CORRECTED-SEARCH() algorithm. We also proposed PRUNED-TRAVERSE() that enhances the performance of CORRECTED-SEARCH() while proving its optimality. Our empirical evaluation results clearly showed the effectiveness of PRUNED-TRAVERSE() both in schedulability and actual runtimes compared to any other existing priority assignment algorithms for PTS.

Notations

τ_i: A task
Γ: The set of tasks {τ_1, τ_2, ..., τ_n}
|Γ|: The number of tasks in task set Γ, the highest priority value
C_i: The worst-case execution time of task τ_i
T_i: The period of task τ_i
D_i: The relative deadline of task τ_i
p_i: The priority of task τ_i
pt_i: The preemption threshold of task τ_i
PO^Γ: The set of all distinct priority orderings {PO_1, PO_2, ..., PO_|Γ|} in task set Γ
PO_n: A priority ordering, a sequence of task priorities ⟨p_i^n, p_2^n, ..., p_2^n⟩
P_n^i: The priority of task τ_i in priority ordering PO_n
PO_A: The resultant priority ordering generated by priority assignment algorithm ALGORITHM()
TO_n: A task ordering from the lowest priority to the highest priority, a sequence of task indexes (i, j, ..., k)
PO_n^{-1}: The inverse mapping of priority ordering PO_n, the task ordering TO_n
TO_n^{-1}: The inverse mapping of task ordering TO_n, the priority ordering PO_n
R_i: The worst-case response time of task τ_i
L_i: The longest level-p_i busy period
B_i: The worst-case blocking time of task τ_i
S_{ij,q}: The start time of the qth instance of task τ_i in L_i
F_{ij,q}: The finish time of the qth instance of task τ_i

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was supported by Hankuk University of Foreign Studies Research Fund of 2015.

References


