Hindawi Publishing Corporation
The Scientific World Journal
Volume 2015, Article ID 837472, 14 pages
http://dx.doi.org/10.1155/2015/837472

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

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 $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_{|\Gamma|}\}$. Each task τ_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 τ_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: every task has a different priority value. Each task set Γ has $|\Gamma|!$ distinct priority orderings for its tasks. Accordingly, the set of distinct priority orderings has cardinality $|\Gamma|!$, which we denote $\mathrm{PO}^{\Gamma} = \{\mathrm{PO}_1, \mathrm{PO}_2, \ldots, \mathrm{PO}_{|\Gamma|!}\}$. 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 Γ, which we denote as $\mathrm{PO}_n = \langle p_1^n, p_2^n, \ldots, p_{|\Gamma|}^n \rangle$. 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 $\mathrm{PO}_n^{-1} = \mathrm{TO}_n = \langle i, j, \ldots, k \rangle$ where each number represents a task index. We also denote the inverse mapping of task ordering TO_n as $\mathrm{TO}_n^{-1} = \mathrm{PO}_n$.

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 $\forall \tau_i$, $\mathrm{pt}_i \geq p_i$. 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 τ_i as follows:

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

$$Q_i = \left\lceil \frac{L_i}{T_i} \right\rceil, \tag{2}$$

$$L_i = B_i + \sum_{\forall j, p_j \geq p_i} \left\lceil \frac{L_i}{T_j} \right\rceil \cdot C_j, \tag{3}$$

Table 1: A walk-through example task set. Bold and italic R_i's represent infeasible response times.

Tasks	C_i	T_i	D_i	DMPO [9]			GREEDY-SA( ) [3]			PA-DMMPT( ) [24]			SEARCH( ) [1]			TRAVERSE( ) [13]		
				p_i	pt_i	R_i	p_i	pt_i	R_i	p_i	pt_i	R_i	p_i	pt_i	R_i	p_i	pt_i	R_i
τ_1	8	43	36	1	4	31	4	4	14	1	4	31	2	4	30	3	3	26
τ_2	4	33	33	2	4	30	2	2	23	3	3	25	3	3	25	2	4	30
τ_3	5	48	31	3	3	26	3	3	19	2	4	30	1	4	31	1	4	31
τ_4	7	14	11	4	4	***14***	1	5	***24***	4	4	***14***	4	4	***14***	4	4	11

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

$$F_{i,q} = S_{i,q} + C_i + \sum_{\forall j, p_j > \mathrm{pt}_i} \left(\left\lceil \frac{F_{i,q}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_{i,q}}{T_j} \right\rfloor \right) \right) \cdot C_j, \quad (4)$$

$$S_{i,q} = B_i + (q-1) \cdot C_i + \sum_{\forall j, p_j \geq p_i} \left(1 + \left\lfloor \frac{S_{i,q}}{T_j} \right\rfloor \right) \cdot C_j, \quad (5)$$

$$B_i = \max \left\{ C_j - 1 \mid \forall j,\ \mathrm{pt}_j \geq p_i > p_j \right\}, \quad (6)$$

where L_i is the longest level-p_i busy period [28], q is the index of instances of task τ_i within L_i, Q_i is the last index of instances of task τ_i within L_i, $F_{i,q}$ is the finish time of the qth instance of task τ_i, $S_{i,q}$ is the start time of the qth instance of task τ_i, and B_i is the worst-case blocking time of task τ_i. Whenever a variable appears on both sides of an equation (i.e., L_i in (3) and $F_{i,q}$ 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 τ_i with the assignment of p_i and pt_i is *feasible* $\leftrightarrow R_i \leq D_i$.

Definition 2 (task set feasibility). Task set Γ with priority ordering $\mathrm{PO}_n = \langle p_1^n, p_2^n, \ldots, p_{|\Gamma|}^n \rangle$ or task ordering $\mathrm{TO}_n = \mathrm{PO}_n^{-1}$ is *feasible* $\leftrightarrow$ every τ_i in Γ is feasible such that τ_i with p_i^n in PO_n 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 $\mathrm{PO}_D = \langle 1, 2, 3, 4 \rangle$. However, as shown in Table 1, this priority ordering makes task τ_4 miss its deadline since $(R_4 = 14) > (D_4 = 11)$. Figure 1(a) demonstrates such a deadline miss: the fifth instance of task τ_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].

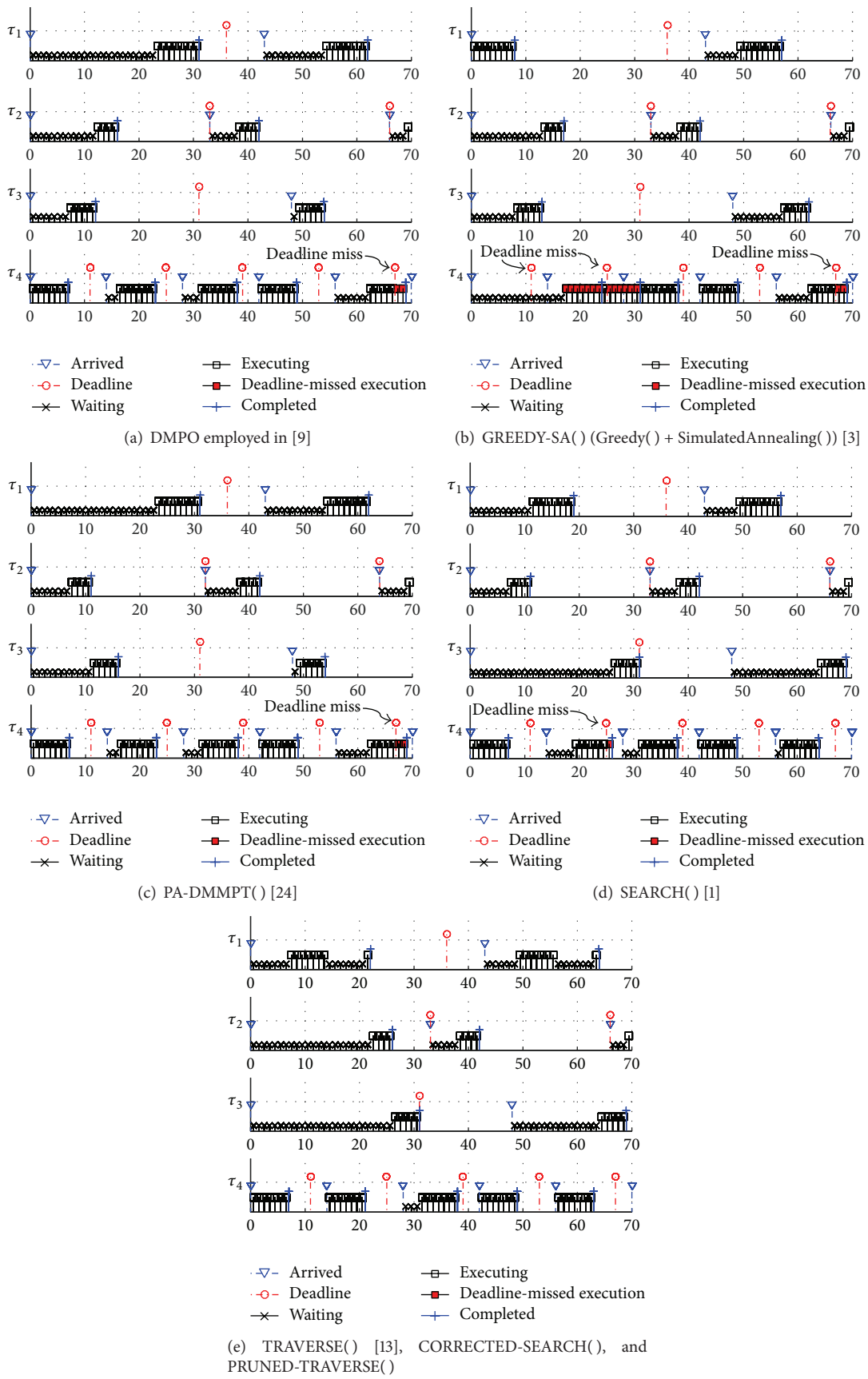
On the other hand, [3, 24] proposed heuristic priority assignment algorithms for PTS. Reference [3] proposed an algorithm that combines Greedy( ) and SimulatedAnnealing( ), which we refer to GREEDY-SA( ) in this paper. Reference [24] proposed PA-DMMPT( ), which means priority assignment algorithm assuming 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, $\mathrm{PO}_G = \langle 4, 2, 3, 1 \rangle$ and $\mathrm{PO}_{\mathrm{PA}} = \langle 1, 3, 2, 4 \rangle$ as shown in Table 1. These priority orderings also make task τ_4 miss its deadline as shown in Table 1.

Figures 1(b) and 1(c) demonstrate such deadline misses of task τ_4. In Figure 1(b), the first, the second, and the fifth instances of task τ_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 τ_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 1(a) shows the pseudo code for TRAVERSE( ). As shown in Algorithm 1(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) Γ, 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



(a) DMPO employed in [9]

(b) GREEDY-SA( ) (Greedy( ) + SimulatedAnnealing( )) [3]

(c) PA-DMMPT( ) [24]

(d) SEARCH( ) [1]

(e) TRAVERSE( ) [13], CORRECTED-SEARCH( ), and PRUNED-TRAVERSE( )

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 τ_4 in (a)~(d) miss their deadlines while every task in (e) does not miss its deadline.

```
(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_i ∈ UnAssigned) {
(6)         p_i ← prio;
(7)         if (_TRAVERSE(prio + 1, UnAssigned − {τ_i}, 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_i ∈ UnAssigned) {
            // WCRT(τ_i, prio): R_i with (p_i ← prio) under fully preemptive fixed priority scheduling
            //                  assuming all tasks in UnAssigned have the highest priority.
(6)         Delay_i ← WCRT(τ_i, prio) − D_i;
(7)         if (Delay_i ≤ 0) {
(8)             p_i ← prio;
(9)             return _SEARCH(prio + 1, UnAssigned − {τ_i}, Γ);
(10)        } // end-if
(11)    } // end-foreach
(12)    SortedList ← ascendingSort(UnAssigned, Delay_i);
(13)    RefinedList ← Refine(SortedList); // eliminating infeasible tasks even with the highest preemption threshold
(14)    foreach (τ_i ∈ RefinedList) {
(15)        p_i ← prio;
(16)        if (_SEARCH(prio + 1, UnAssigned − {τ_i}, Γ) = success) return success;
(17)    } // end-foreach
(18)    return fail;
(b) SEARCH( ) [1]
```

Algorithm 1: Pseudo code for (a) TRAVERSE( ) [13] and (b) SEARCH( ) [1].

this, returning at the last lines (line (9)) happens when a specific priority ordering is not feasible.

The remaining part is composed of a loop that recursively invokes _TRAVERSE( ): each task in *UnAssigned* at line (5) is assigned the priority *prio* at line (6) and remaining unassigned tasks ($UnAssigned - \{\tau_i\}$) are recursively assigned $prio + 1$ at line (7). Note that the recursive invocation of _TRAVERSE( ) returns only when the return value of the recursive invocation is *success* (line (7)). We call this priority assignment as the *tentative* priority assignment since if the return value of the recursion is *fail*, the next task in *UnAssigned* is tried for assigning priority *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. Each number with round braces "(n)" besides a node represents nth feasibility test performed for the operation of the node.

In Figure 2(a), 15 complete task orderings $TO_1 = \langle 1, 2, 3, 4\rangle$, $TO_2 = \langle 1, 2, 4, 3\rangle, \ldots, TO_{15} = \langle 3, 2, 1, 4\rangle$ were generated. The last task ordering $TO_{15} = TO_T = \langle 3, 2, 1, 4\rangle$ has a dangled black triangle node, which indicates a feasible preemption threshold assignment. Task ordering TO_T corresponds to the priority ordering $PO_T = TO_T^{-1} = \langle 3, 2, 1, 4\rangle$ 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^{Γ} until it finds a feasible priority ordering.

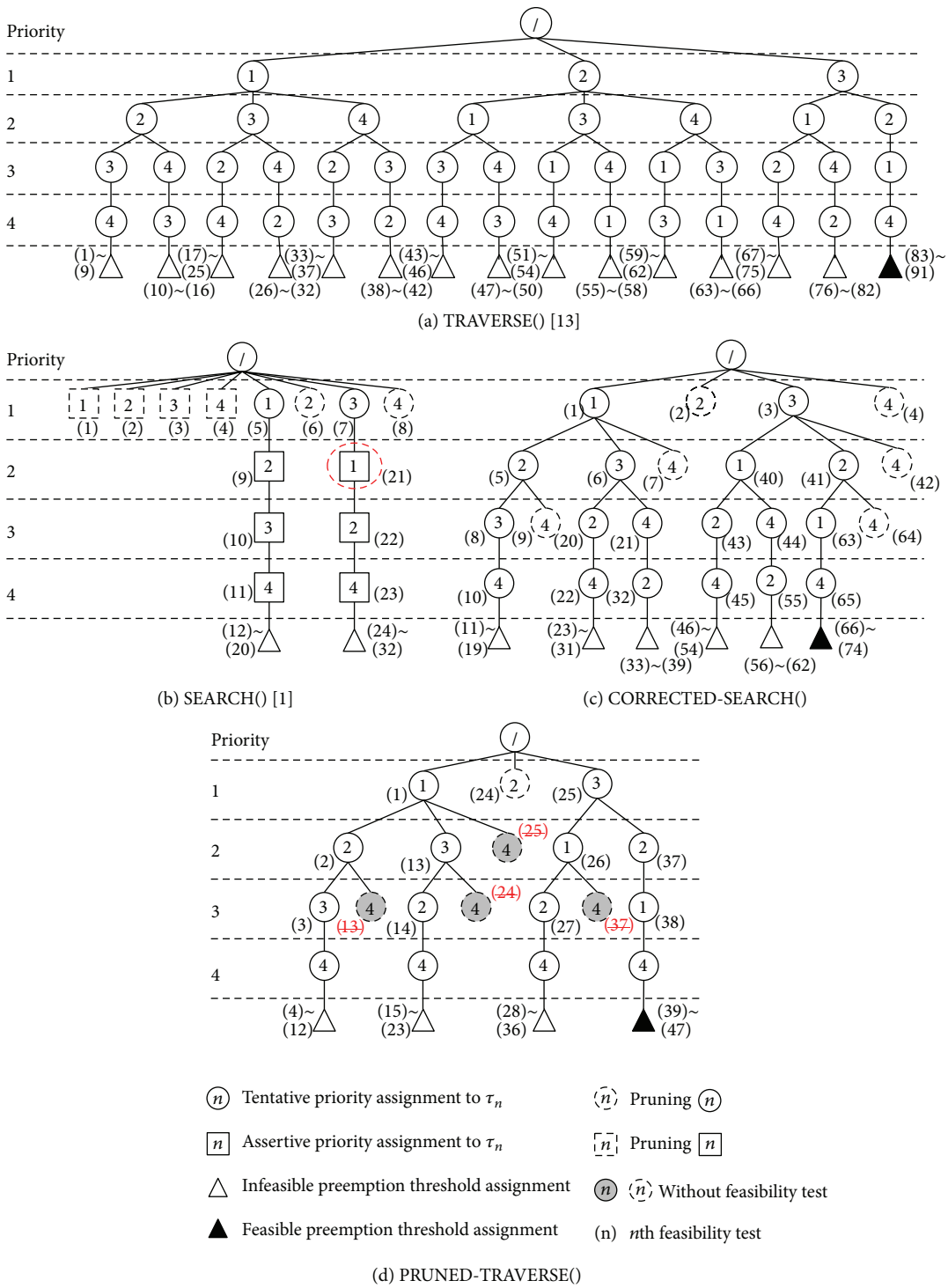


Figure 2: Priority assignment trees for the walk-through example task set of Table 1.

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 $|\Gamma|!$ tentative priority orderings in the worst-case, each of which requires maximally $|\Gamma|^2$ 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 *RefinedList* with *UnAssigned* at line (14). The additional part of SEARCH( ) is composed of one loop (lines (5)~(11)) and refining *UnAssigned* to prepare *RefinedList* (lines (12) and (13)). In contrast to the priority assignment loop of TRAVERSE( ), this additional first loop in SEARCH( ) assigns priority *prio* to task τ_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 τ_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 *RefinedList* instead of *UnAssigned*. The *RefinedList* 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: $\langle 1, 2, 3, 4\rangle$ and $\langle 3, 1, 2, 4\rangle$. Both orderings failed to assign feasible preemption thresholds to tasks as the dangled white triangle nodes indicate. The last task ordering $\mathrm{TO}_S = \langle 3, 1, 2, 4\rangle$ corresponds to the priority ordering $\mathrm{PO}_S = \mathrm{TO}_S^{-1} = \langle 2, 3, 1, 4\rangle$ for SEARCH( ) of Table 1. As shown in Table 1, SEARCH( ) makes task τ_4 miss its deadline since $(R_4 = 14) > (D_4 = 11)$. Figure 1(d) demonstrates such a deadline miss of task τ_4 where the second instance of task τ_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 $\mathrm{PO}_T = \langle 3, 2, 1, 4\rangle$ 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 $\mathrm{TO}_S = \langle 3, 1, 2, 4\rangle$ is the 13th task ordering generated by TRAVERSE( ). The reason why SEARCH( ) failed is due to the assertive priority assignment of $p_1^S \leftarrow 2$ 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_i of task τ_i is fixed, its worst-case response time R_i does not decrease when its preemption threshold pt_i is lowered.*

Proof. While R_i is calculated from (1)~(6), pt_i is only included in (4) for calculating $F_{i,q}$, specifically in the last term of the right side of (4): $\sum_{\forall j, p_j > \mathrm{pt}_i}(\lceil F_{i,q}/T_j\rceil - (1 + \lfloor S_{i,q}/T_j\rfloor)) \cdot C_j$. We refer to this term as $I_{i,q}^F$ for the convenience of proving. $I_{i,q}^F$ is the summation of the interference time by task τ_j such that $\forall j$, $p_j > \mathrm{pt}_i$ after $S_{i,q}$. Accordingly, the set of tasks that can interfere task τ_i with the lower pt_i is always the superset of the set of such tasks with the higher pt_i value. Consequently, $I_{i,q}^F$ does not decrease with lower pt_i. From (1), R_i monotonically increases with respect to to $F_{i,q}$ and $F_{i,q}$ is proportional to $I_{i,q}^F$ from (4). Therefore, R_i is also proportional to $I_{i,q}^F$ and thus R_i does not decrease with lower pt_i. □

Theorem 4. *Under PTS, if task τ_i with priority p_i^n in priority ordering PO_n and the highest preemption threshold pt_i ($\mathrm{pt}_i = |\Gamma|$) is infeasible, task set Γ with PO_n is also infeasible.*

Proof. From Definition 1, task τ_i is feasible if and only if $R_i \leq D_i$. From Lemma 3, if p_i is fixed (as p_i^n), $\mathrm{pt}_i = |\Gamma|$ guarantees the minimal R_i for τ_i. Accordingly, if task τ_i with fixed p_i and $\mathrm{pt}_i = |\Gamma|$ is infeasible, no other preemption threshold assignment can make R_i reduced, that is, τ_i feasible. From Definition 2, Γ with PO_n is feasible if and only if all tasks with PO_n in Γ 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:

(i) deletion of the first loop of the assertive priority assignment and sorting tasks (lines (5)~(12) of Algorithm 1(b)),

(ii) replacement of *SortedList* with *UnAssigned* (line (13) of Algorithm 1(b)),

(iii) adding "$\mathrm{pt}_i \leftarrow prio$;" after "$p_i \leftarrow prio$;" (line (15) of Algorithm 1(b)),

(iv) replacement of OPT-ASSIGN-THRESHOLD(Γ) at line (4) with RESTORING-OPT-ASSIGN-THRESHOLD(Γ) 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 *RefinedList* instead of *UnAssigned*. From Theorem 4, pruning at line (5) that makes *RefinedList*

```
(1) CORRECTED-SEARCH (Γ: set of tasks)
(2)     return _CORRECTED-SEARCH(1, Γ, Γ);

(3) _CORRECTED-SEARCH (prio: priority, UnAssigned: set of tasks, Γ: set of tasks)
(4)  if (UnAssigned = { }) return RESTORING-OPT-ASSIGN-THRESHOLD(Γ);
(5)  RefinedList ← Refine(UnAssigned); // eliminating infeasible tasks even with the highest preemption threshold
(6)  foreach (τ_i ∈ RefinedList) {
(7)      p_i ← prio;
(8)      pt_i ← prio;
(9)      if (_SEARCH(prio + 1, UnAssigned − {τ_i}, Γ) = success) return success;
(10)    } // end-foreach
(11)    return fail;
(a) CORRECTED-SEARCH( )

(1) RESTORING-OPT-ASSIGN-THRESHOLD (Γ: set of tasks)
(2)     ret ← OPT-ASSIGN-THRESHOLD(Γ);
(3)     if (ret = fail) foreach (τ_i ∈ Γ) pt_i ← p_i; // restore preemption thresholds
(4)     return ret;
(b) RESTORING-OPT-ASSIGN-THRESHOLD( )
```

Algorithm 2: Pseudo code for (a) CORRECTED-SEARCH( ) and (b) RESTORING-OPT-ASSIGN-THRESHOLD( ).

```
(1) PRUNED-TRAVERSE (Γ: set of tasks)
(2)     Γ ← descendingSort(Γ, D_i);
(3)     foreach (τ_i ∈ Γ) {
(4)         infeasiblePrioMax_i ← 0;
(5)         p_i ← |Γ|;
(6)         pt_i ← |Γ|;
(7)     } // end-foreach
(8)     return_PRUNED-TRAVERSE(1, Γ, Γ);

(9) _PRUNED-TRAVERSE (prio: priority, UnAssigned: set of tasks, Γ: set of tasks)
(10)    foreach (τ_i ∈ UnAssigned) {
(11)        if (prio < infeasiblePrioMax_i) continue; // pruning this path with condition C2
(12)        p_i ← prio;
(13)        pt_i ← |Γ|;
(14)        if (prio = |Γ|) {
(15)            ret = RESTORING-OPT-ASSIGN-THRESHOLD(Γ);
(16)            if (ret = fail) infeasiblePrioMax_i ← |Γ|;
(17)            return ret;
(18)        } // end-if
(19)        if (R_i > D_i) {
(20)            p_i ← |Γ|; // restore to the highest priority
(21)            if (prio > infeasiblePrioMax_i) infeasiblePrioMax_i ← prio;
(22)            continue; // pruning this path with condition C1
(23)        } // end-if
(24)        pt_i ← prio;
(25)        if (_PRUNED-TRAVERSE(prio + 1, UnAssigned − {τ_i}, Γ) = success) return success;
            // The following line is executed when the above _PRUNED-TRAVERSE( ) returned fail.
(26)        foreach (τ_j ∈ UnAssigned) p_j ← |Γ|; // restore to the highest priority
(27)    } // end-foreach
(28)    return fail;
```

Algorithm 3: Pseudo code for the proposed PRUNED-TRAVERSE( ) algorithm.

is valid. With this, we can easily see that CORRECTED-SEARCH( ) is optimal since TRAVERSE( ) is optimal.

Third, we assign task τ_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 $\Gamma - UnAssigned$. 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 $\mathrm{TO}_C = \langle 3, 2, 1, 4\rangle$ is the same as TO_T, and the resultant feasible task ordering of TRAVERSE( ). CORRECTED-SEARCH( ) requires 74 feasibility tests while TRAVERSE( ) requires 91 feasibility tests as the numbers besides 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 p_i of a task τ_i is lowered to p_i' ($p_i' < p_i$), there exists at least one lower priority task τ_j ($p_j < p_i$) that heightens its priority p_j to p_j' with $p_j' > p_i'$.*

Proof. The lemma can be easily proved by observing task orderings in priority assignment trees like Figure 2(a). If task τ_i 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_i of task τ_i is fixed, its worst-case response time R_i does not decrease when its priority p_i is lowered.*

Proof. For the convenience of proving, we refer to the last terms of the right sides of (4) and (5) as $I_{i,q}^F$ and $I_{i,q}^S$, respectively. We prove the theorem by contradiction. Suppose that R_i of task τ_i decreases to R_i' when its priority p_i is lowered to p_i': $R_i' < R_i$ and $p_i' < p_i$. For R_i to decrease, $F_{i,q}$ should decrease from (1). For $F_{i,q}$ to decrease, $S_{i,q}$ or $I_{i,q}^F$ should decrease from (4). Since the condition for calculating $I_{i,q}^F$ is $p_j > \mathrm{pt}_i$, the change of priority p_i does not affect $I_{i,q}^F$. Therefore, $S_{i,q}$ should decrease to $S_{i,q}'$ such that $S_{i,q} - S_{i,q}' > 0$. Then, from (5), $(B_i + I_{i,q}^S) - (B_i' + I_{i,q}^{S\,\prime}) > 0$ follows where B_i' and $I_{i,q}^{S\,\prime}$ are changed values of B_i and $I_{i,q}^S$ due to the lower priority. Then, $(B_i - B_i') > (I_{i,q}^{S\,\prime} - I_{i,q}^S)$ follows.

From Lemma 5, if p_i is lowered to p_i' there exists at least one lower priority task τ_j ($p_j < p_i$) that heightens its priority p_j to p_j' such that $p_j' > p_i'$. Let the set of such additionally introduced higher priority tasks after the priority of task τ_i is lowered be $AddedHP_i = \{\tau_j \mid p_j < p_i \text{ and } p_j' > p_i'\}$. The maximum possible value of $(B_i - B_i')$ is achieved when $B_i' = 0$, which infers that at least one task in $AddedHP_i$ contributes in making B_i. Let such a task in $AddedHP_i$ be task τ_k; that is $B_i = C_k$. Then, $\max(B_i - B_i') = C_k$ follows. On the other hand, from (5), it follows that $I_{i,q}^{S\,\prime} - I_{i,q}^S = \sum_{\forall j, p_j \in AddedHP_i} (1 + \lfloor S_{i,q}'/T_j \rfloor) \cdot C_j \geq C_k$. Then, $(B_i - B_i') \leq (I_{i,q}^{S\,\prime} - I_{i,q}^S)$ follows. This is a contradiction, which proves the theorem. □

Theorem 7. *Under PTS, if task τ_i with priority p_i^n in priority ordering PO_n and the highest preemption threshold $\mathrm{pt}_i = |\Gamma|$ is infeasible, task set Γ with another priority ordering PO_m that assigns task τ_i the lower priority p_i^m (such that $p_i^m < p_i^n$) is also infeasible.*

Proof. Let the worst-case response time of task τ_i with $\mathrm{pt}_i = |\Gamma|$ and p_i^n be R_i. Let the worst-case response time of task τ_i with $\mathrm{pt}_i = |\Gamma|$ and p_i^m be R_i'. From Lemma 3, both R_i and R_i' are the minimal possible worst-case response time with each given priority. From Theorem 6, $R_i' \geq R_i$ follows. Since task τ_i with p_i^n and $\mathrm{pt}_i = |\Gamma|$ is infeasible, $R_i > D_i$ follows from Definition 1. Therefore, $R_i' > D_i$ follows. Accordingly, task τ_i with p_i^m and $\mathrm{pt}_i = |\Gamma|$ is infeasible. Consequently, from Theorem 4, task set Γ with PO_m is also infeasible. □

Now we propose PRUNED-TRAVERSE( ) that extends TRAVERSE( ) by exhaustively pruning infeasible paths. PRUNED-TRAVERSE( ) prunes such priority ordering PO_n that assigns task τ_i priority p_i^n when the following condition is true:

$$\mathrm{C1} \vee \mathrm{C2}, \tag{7}$$

where C1 is $\mathrm{pt}_i = |\Gamma| \wedge R_i > D_i$, and C2 is $p_i^n < \max\{p_i \mid \mathrm{C1}\}$.

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( ), PRUNED-TRAVERSE( ) depends on its subroutine, _PRUNED-TRAVERSE( ), which has the same parameters as _TRAVERSE( ): *prio*, *UnAssigned*, and Γ (line (9)). Like _TRAVERSE( ), _PRUNED-TRAVERSE( ) assigns priorities to tasks from the lowest priority 1 to the highest priority $|\Gamma|$: it assigns each task τ_i in *UnAssigned* (line (10)) priority *prio* (line (12)) and the

remaining unassigned tasks ($UnAssigned - \{\tau_i\}$) are recursively assigned priority $prio + 1$ at line (21).

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

Unlike _TRAVERSE( ), _PRUNED-TRAVERSE( ) needs to perform a feasibility test for task τ_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 *UnAssigned* have higher priorities than the priorities of the previously priority assigned tasks. For this, PRUNED-TRAVERSE( ) assigns the highest priority $|\Gamma|$ to all unassigned tasks initially at line (5). _PRUNED-TRAVERSE( ) also does so conditionally at line (26) whenever the next priority ($prio + 1$) 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 τ_i priority *prio* (line (12)), _PRUNED-TRAVERSE( ) first prunes any infeasible priority ordering at line (11) if *prio* is smaller than $infeasiblePrioMax_i$, 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 $infeasiblePrioMax_i$. If task τ_i is assigned priority *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 *prio* of the maximum priority $|\Gamma|$ is assigned (line (14)), which means that a complete priority ordering is generated, _PRUNED-TRAVERSE( ) assigns preemption thresholds by invoking RESTORING-OPT-ASSIGN-THRESHOLD( ) in Algorithm 2(b). Note that there are two differences with TRAVERSE( ) in assigning preemption thresholds: (1) invoking RESTORING-OPT-ASSIGN-THRESHOLD( ) instead of OPT-ASSIGN-THRESHOLD( ) and (2) assigning preemption thresholds to tasks once the maximum priority is assigned instead once *UnAssinged* is $\{\}$. (1) is for the proper pruning operation as we explained in proposing 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 PRUNED-TRAVERSE( ) compared to TRAVERSE( ) or CORRECTED-SEARCH( ).

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

_PRUNED-TRAVERSE( ) requires careful restorations of priorities and preemption thresholds of tasks, which are repetitively and tentatively assigned. Since SEARCH( ) and CORRECTED-SEARCH( ) prune infeasible paths within subroutine Refine( ), such restoration is easier and less error-prone. However, 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 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 PRUNED-TRAVERSE( ).

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

Proof. PRUNED-TRAVERSE( ) extends 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 TRAVERSE( ) is optimal, PRUNED-TRAVERSE( ) is optimal. □

Figure 2(d) shows the priority assignment tree of 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 PRUNED-TRAVERSE( ) instead of using subroutine Refine( ) of 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 $\text{TO}_P = \langle 3, 2, 1, 4 \rangle$ is the same as TO_C and TO_T, which are the resultant feasible task ordering of CORRECTED-SEARCH( ) and TRAVERSE( ), respectively. PRUNED-TRAVERSE( ) produces four complete priority orderings and requires 47 feasibility tests while CORRECTED-SEARCH( ) produces six complete priority orderings and requires 74 feasibility tests. As such, it is obvious that the overall performance of PRUNED-TRAVERSE( ) is much better than that of CORRECTED-SEARCH( ) since PRUNED-TRAVERSE( ) prunes more infeasible paths than 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 PO^{Γ}, each of which requires $O(n^2)$ feasibility tests [3] via OPT-ASSIGN-THRESHOLD( ). With this, TRAVERSE( ) has the complexity of $O(E \cdot n! \cdot n^2)$.

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 $O(E \cdot n! \cdot n^2) + O(E \cdot n!) = O(E \cdot n! \cdot n^2)$, 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 be 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 $U_B = n \cdot (\sqrt[n]{2} - 1)$. For example, if $|\Gamma| = 10$ and $U \leq 0.72$ ($\cong 10 \cdot (\sqrt[10]{2} - 1) = U_B$), 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 τ_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.

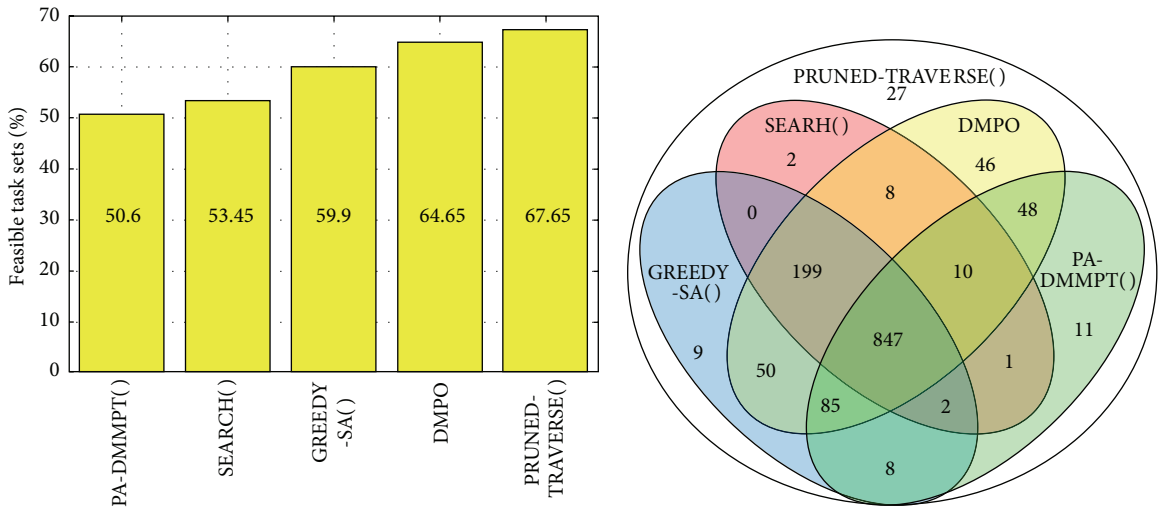
Figure 3(a) clearly shows that the schedulability performance results of PA-DMMPT( ) [24], SEARCH( ) [1], GREEDY-SA( ) [3], DMPO [9], and PRUNED-TRAVERSE( ) are uniformly improved in order. Specifically, PA-DMMPT( ), SEARCH( ), GREEDY-SA( ), DMPO, and PRUNED-TRAVERSE( ) could, respectively, schedule 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.

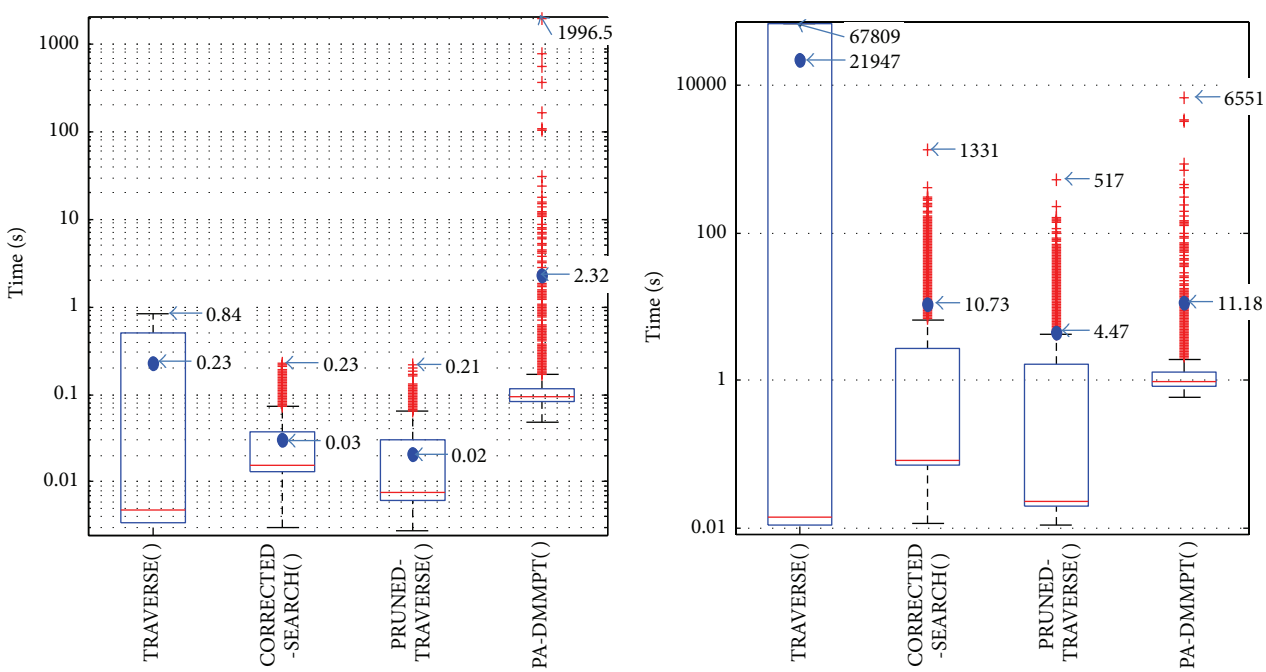
Figures 4(a) and 4(b) clearly show that the actual runtime performance distribution results of TRAVERSE( ), CORRECTED-SEARCH( ), and PRUNED-TRAVERSE( ) are



(a) Ratio of feasible task sets in percentage

(b) Venn diagram for the number of feasible task sets in 2,000 task sets

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



(a) $|\Gamma| = 5$

(b) $|\Gamma| = 10$

Figure 4: Actual runtimes in seconds when $U = 0.9$.

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

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 $|\Gamma| = 10$, 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 $|\Gamma| = 20$, 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 than 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 $O(n^2)$ 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 $\{\tau_1, \tau_2, \ldots, \tau_{|\Gamma|}\}$
$|\Gamma|$: 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 $\{\mathrm{PO}_1, \mathrm{PO}_2, \ldots, \mathrm{PO}_{|\Gamma|!}\}$ in task set Γ
PO_n: A priority ordering, a sequence of task priorities $\langle p_1^n, p_2^n, \ldots, p_{|\Gamma|}^n\rangle$
p_i^n: 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 $\langle i, j, \ldots, k\rangle$
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
$S_{i,q}$: The start time of the qth instance of task τ_i in L_i
$F_{i,q}$: The finish time of the qth instance of task τ_i in L_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

[1] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, pp. 328–335, IEEE, Hong Kong, China, 1999.

[2] R. Ghattas and A. G. Dean, "Preemption threshold scheduling: stack optimality, enhancements and analysis," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, pp. 147–157, IEEE, Bellevue, Wash, USA, April 2007.

[3] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pp. 25–34, IEEE, Orlando, Fla, USA, November 2000.

[4] Y. Ge, Y. Dong, and H. Zhao, "Energy-efficient task scheduling and task energy consumption analysis for real-time embedded systems," in *Proceedings of the Theoretical Aspects of Software Engineering (TASE '14)*, pp. 135–138, IEEE, Changsha, China, September 2014.

[5] Q. Zhao, Z. Gu, and H. Zeng, "PT-AMC: integrating preemption thresholds into mixed-criticality scheduling," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '13)*, pp. 141–146, IEEE, Grenoble, France, March 2013.

[6] L. Yang and L. Man, "On-line and off-line DVS for fixed priority with preemption threshold scheduling," in *Proceedings of International Conference on Embedded Software and Systems (ICESS '09)*, pp. 273–280, IEEE, Zhejiang, China, May 2009.

[7] OSEK/VDX, http://www.osek-vdx.org/.

[8] AUTOSAR (Automotive Open System Architecture, http://www.autosar.org/.

[9] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. A survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[10] ExpressLogic, Preemption Threshold Scheduling in ThreadX Real-Time Operating System (RTOS), http://rtos.com/preemption-threshold_scheduling/.

[11] J. Carbone, "Cutting overhead through preemption threshold scheduling," New Electronics, 2013, http://www.newelectronics.co.uk/electronics-technology/cutting-overhead-through-preemption-threshold-scheduling/56460/.

[12] A. G. Dean, "Lower the overhead in RTOS scheduling," *Embedded Systems Design*, vol. 24, no. 2, 2011.

[13] S. Kim, "Synthesizing multithreaded code from real-time object-oriented models via schedulability-aware thread derivation," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 413–426, 2014.

[14] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

[15] S. Kim, J. Park, and S. Hong, "Scenario-based multitasking for real-time object-oriented models," *Information and Software Technology*, vol. 48, no. 9, pp. 820–835, 2006.

[16] B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Addison-Wesley, Reading, Mass, USA, 1999.

[17] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*, Addison-Wesley, 2004.

[18] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, Longman, 2000.

[19] OMG, *Unified Modeling Language (UML) 2.0*, 2007, http://www.uml.org/.

[20] N. C. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[21] K. Bletsas and N. Audsley, "Optimal priority assignment in the presence of blocking," *Information Processing Letters*, vol. 99, no. 3, pp. 83–86, 2006.

[22] A. Zuhily and A. Burns, "Optimal $(D - J)$-monotonic priority assignment," *Information Processing Letters*, vol. 103, no. 6, pp. 247–250, 2007.

[23] R. J. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Behnam, "Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds," in *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS '14)*, pp. 161–172, Rome, Italy, December 2014.

[24] H. Zeng, M. D. Natale, and Q. Zhu, "Minimizing stack and communication memory usage in real-time embedded applications," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 5, supplement, article 149, 25 pages, 2014.

[25] J. Chen, A. Harji, and P. Buhr, "Solution space for fixed-priority with preemption threshold," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 385–394, IEEE, March 2005.

[26] U. G. Keskin, R. J. Bril, and J. J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '10)*, pp. 1–4, Bilbao, Spain, September 2010.

[27] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 315–326, IEEE, December 2002.

[28] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS '90)*, pp. 201–209, IEEE, December 1990.

[29] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[30] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[31] R. I. Davis, A. Zabos, and A. Burns, "Efficient exact schedulability tests for fixed priority real-time systems," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1261–1276, 2008.