JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 319-335 (2018)
DOI: 10.6688/JISE.201803_34(2).0001

Efficient Optimal Priority Assignment for Fixed Priority Preemption Threshold Scheduling*

SAEHWA KIM
Department of Information Communications Engineering
Hankuk University of Foreign Studies
Gyeonggi-do, 449-791 Korea
E-mail: ksaehwa@hufs.ac.kr

This paper proposes an efficient priority assignment algorithm for fixed priority preemption threshold scheduling (PTS), which we named FAST-TRAVERSE. It is optimal in the sense that it always finds a feasible priority assignment if one exists. While there are existing optimal algorithms, they are inefficient to be used in practice. The key ideas of FAST-TRAVERSE are to prune sibling traverses and preemption threshold assignments if possible based on the notion of the effective blocking task. The empirical evaluation results clearly show that FAST-TRAVERSE achieves the largest feasibility and can be employed as an on-line priority assignment algorithm for PTS.

Keywords: real-time systems and embedded systems, system integration and implementation, real-time feasibility, scalability, fixed-priority scheduling

1. INTRODUCTION

Fixed priority preemption threshold scheduling (PTS) [1] has been widely applied in the real-time industry due to its effectiveness and simplicity. PTS is an extension of preemptive fixed priority scheduling where each task has a preemption threshold as a scheduling attribute in addition to its priority. The preemption threshold of a task is a form of run-time priority that remains after the task has been dispatched and until its execution is completed, so it regulates the degree of "preemptiveness" in fixed priority scheduling. If the threshold of each task is the same as the original priority, then PTS becomes equivalent to fully-preemptive fixed priority scheduling. If the threshold of each task is the highest priority in the system, then PTS becomes equivalent to non-preemptive scheduling (NPS). The use of PTS is very effective in system tuning processes since it improves real-time schedulability, eliminates unnecessary preemptions, reduces the number of tasks since a group of non-preemptive tasks can be considered to be a single task, and allows for scalable real-time systems to be designed [2].

To increase the real-time schedulability (feasibility) via the adoption of PTS, we need an algorithm that assigns to each task feasible scheduling attributes, that is, the priority and the preemption threshold. A scheduling attributes assignment algorithm is *optimal* if it is guaranteed to output a feasible (schedulable) scheduling attributes assignment if one exists [3-6]. While there are previously proposed optimal scheduling attributes assignment algorithms for PTS [7, 8], they are very inefficient to be used in practice

Received July 11, 2016; revised September 17, 2016; accept November 14, 2016.
Communicated by Shao-Li Tsao.
* This work was supported by Hankuk University of Foreign Studies Research Fund of 2017. This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2017R1A2B1001824).

when the number of tasks are more than ten: it is impossible to get the assignment result even with modern computers. With this, some research activities have proposed heuristic scheduling attributes assignment algorithms [1, 9, 10] for PTS. However, they often fail to find feasible scheduling attributes since they are non-optimal, and such practical needs have motivated our work.

This paper proposes an efficient optimal priority assignment algorithm for fixed priority preemption threshold scheduling (PTS), which we named FAST-TRAVERSE. It is optimal in the sense that it is guaranteed to find a feasible priority assignment if one exists. While FAST-TRAVERSE is an extension of PRUNED-TRAVERSE [8], the former is dramatically faster the latter and even faster than PA-DMMPT [10], which is the best heuristic algorithm to the best of our knowledge. Our empirical results show that the average algorithm run time of FAST-TRAVERSE when the number of tasks is 50 was only 49% larger than that of DMPO (Deadline Monotonic Priority Ordering), which is the mostly widely used and one of the simplest assignment algorithm. With this, FAST-TRAVERSE can be employed as an on-line priority assignment algorithm for PTS. For example, it can be employed for the on-line admission control system that decides whether newly arrived tasks are acceptable or not.

As all the other priority assignment algorithms for PTS, FAST-TRAVERSE assigns priorities to tasks from the lowest priority to the highest priority, whose appropriateness with respects to the efficiency of the feasibility test is widely known, as shown in [11]. If one priority ordering is found to be infeasible, FAST-TRAVERSE traverses another priority ordering also from the lowest unassigned priority by a sibling traverse. Here a sibling traverse means assigning the same priority to the remaining tasks in the unassigned task set.

Our key idea is to prune sibling traverses and preemption threshold assignments if possible. Once a priority ordering is found to be infeasible, FAST-TRAVERSE finds the minimum priority of the priority ordering that causes the priority ordering to be infeasible. For this, we try to find the *effective* blocking task that actually causes the task set infeasible. Once we find such a task, we also set the minimum infeasible preemption threshold value of the task. With this, we prune any preemption threshold assignment that assigns preemption threshold larger than that value.

The remainder of the paper is organized as follows. In the next subsection, we present the related work. Section 2 presents the task model with some notations and definitions. Section 3 discusses previous algorithms with a walk-through example. Section 4 specifies the proposed algorithm, FAST-TRAVERSE, with the theoretical backgrounds. Section 5 gives empirical performance evaluation results. Finally, Section 6 concludes the paper.

1.1 Related Work

There are various previously proposed priority and preemption threshold assignment algorithms for PTS. To begin with, DMPO (deadline monotonic decreasing order) is optimal in the fully-preemptive fixed priority scheduling [6] and is so even though there are blockings if there is no jitter [5]. Therefore, the approach of assigning priorities using DMPO and then assigning preemption thresholds using the optimal preemption threshold assignment algorithm of [4], which we refer to OPT-ASSIGN-THRESHOLD, is widely

used in practice. In this paper, DMPO means deadline-monotonic priority assignment combined with OPT-ASSIGN-THRESHOLD, as this approach was employed in [9, 12].

However, DMPO is not optimal algorithm in the sense that even if a task set is infeasible with DMPO, the task set may be indeed feasible with another priority assignment. Therefore, optimal algorithms such as SEARCH [4][1], TRAVERSE [7], CORRECTED-SEARCH [8], and PRUNED-TRAVERSE [8] have been proposed. PRUNED-TRAVERSE [8] is the most efficient (fast) one among all existing optimal algorithms. However, its complexity is $O(n!\cdot n^2)$ where n is the number of tasks in a task set, which means computationally intractable if n is large. For example, PRUNED-TRAVERSE cannot give an assignment result for some task set in a single day even in a high-performance modern computer when n is more than 15. Therefore, existing optimal algorithms cannot be employed in practice, which motivated our work.

Since optimal algorithms are too inefficient, heuristic algorithms such as GREEDY [1] and PA-DMMPT [10] were proposed. Among these heuristic algorithms, PADMMPT is the best with respect to the possibility of finding a feasible priority assignment. However, it is not optimal and thus may not find a feasible priority assignment for a given task set, which will be demonstrated in Section 3.

On the other hand, [12] proposed an optimal preemption threshold assignment algorithm that considers the cash related preemption delay (CRPD) for PTS. However, it assumes pre-assigned priorities for a given task set and thus still does not guarantee an optimal priority assignment result for a given task set.

2. TASK MODEL

We use the same task model as the one used in the traditional fixed-priority preemption threshold scheduling [1, 4, 13, 14]. We assume a uniprocessor system and 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. All timing values are real numbers. Each task τ_i also has a fixed priority p_i and a preemption threshold pt_i where p_i and pt_i are assigned by a specific priority and preemption threshold assignment algorithm. 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 [4]: which means that $\forall \tau_i, pt_i \geq p_i$.

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. We denote the resultant priority ordering generated by a specific priority assignment algorithm ALGORITHM as PO_{AL}. With this, a specific priority ordering PO_n is a sequence of priorities for tasks in task set Γ, which we denote as $PO_n = \langle p^n_1, p^n_2, \ldots, p^n_{|\Gamma|}\rangle$. 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 $TO_n^{-1} = PO_n$. Besides, a specific preemption threshold ordering is a sequence of preemption thresholds for tasks in task set Γ, which we denote as $PTO_n = \langle pt^n_1, pt^n_2, \ldots, pt^n_{|\Gamma|}\rangle$.

[1] SEARCH is not optimal in fact [8].

As the feasibility test under PTS, we adopt the worst-case response time analysis equations of [9] while extending its integer time model to real number time model. Original equations were introduced by [4], whose errors were fixed by [15]. These results were refined by [14], 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.

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

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

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

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

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

$$B_i = \max \left\{ C_j - \varepsilon | \forall j, pt_j \geq p_i > p_j \right\}, \tag{6}$$

where L_i is the longest level–p_i busy period [16], 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, B_i is the worst-case blocking time of task τ_i, and ε is the infinitesimal amount of time larger than zero. Whenever a variable appears on both sides of the Eq. (*i.e.* L_i in Eq. (2) and $F_{i,q}$ in Eqs. (3) and (4)), its value can be found by iterating until the value converges [15]. Refer to [9] for the appropriate initial values for the iterations.

Note that ε in Eq. (6) was 1 in [9] that used the integer time model where all real numbers of timing parameters should be adapted to integer numbers. The value of ε should be set as the upper bound on the relative error due to rounding in real numbers. The original analysis in [14] that the equations of [9] are based on clearly shows the validity of using ε instead of 1. By using ε instead of 1 in calculating blocking time B_i, we allow real numbers in timing parameters such as C_i.

Note also that Eqs. (1)-(6) provide tight worst-case response time results and not an upper bound on the worst-case response time such as [17], which can be used just for the sufficient feasibility test. In other words, these equations provides an "exact" worst-case response time under PTS as proved in [14], and thus can be used for the exact feasibility test. Based on this exact worst-case response time analysis, this paper proposes an efficient optimal priority and preemption threshold assignment algorithm.

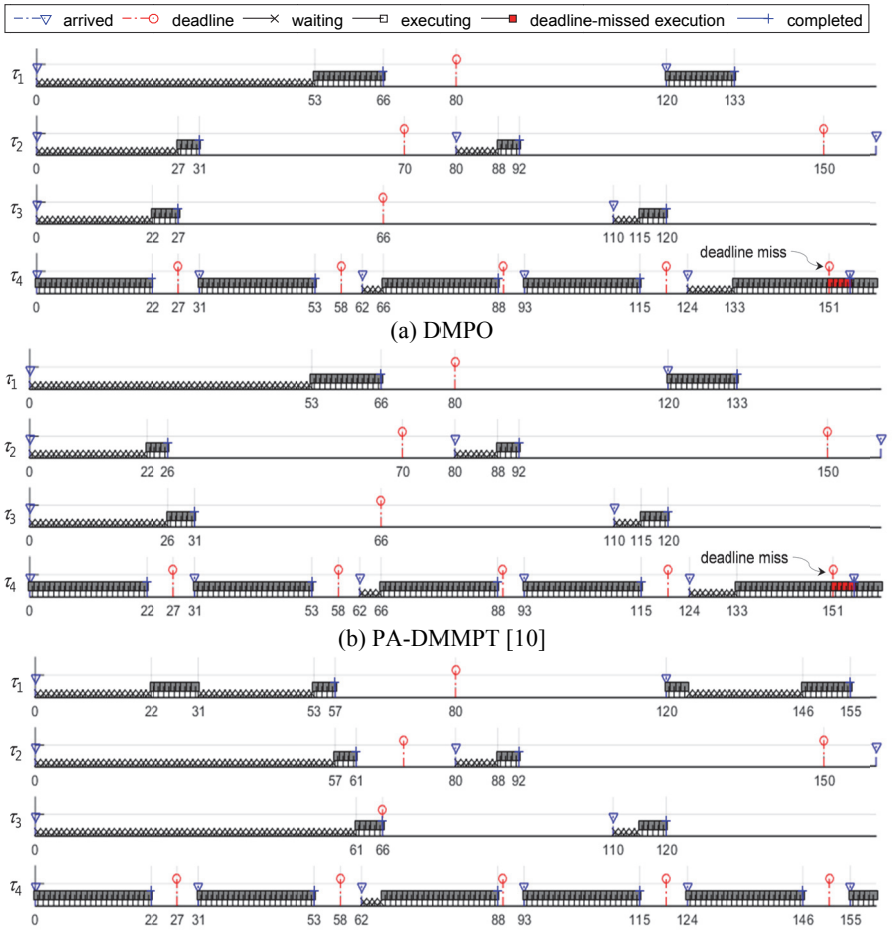
2.1 Walk-through Example Task Set

As a walk-through example task set, we use a task set in Table 1 that is composed of four tasks. The task indices of the task set are in the deadline monotonic decreasing order (DMPO). For this task set, the resultant priority ordering of DMPO is $PO_{DM} = \langle 1, 2, 3, 4 \rangle$.

However, as shown in Table 1, this priority ordering makes task τ_4 miss its deadline since $(R_4 = 35) > (D_4 = 27)$. Fig. 1 (a) demonstrates such a deadline miss: the fifth instance of task τ_4 completes at time point 155 while its absolute deadline is $(5 - 1) \cdot T_4 + D_4 = 151$. Note that the worst-case response time cannot be obtained at the critical instant [18] when there is a non-preemptiveness of tasks [9].

Table 1. Walk-through example task set.

	C_i	T_i	D_i	DMPO			PA-DMMPT			Optimal Assignment		
				p_i	pt_i	R_i	p_i	pt_i	R_i	p_i	pt_i	R_i
τ_1	13	120	80	1	4	66	1	4	66	3	3	62
τ_2	4	80	70	2	4	66	3	3	61	2	4	66
τ_3	5	110	66	3	3	62	2	4	66	1	4	66
τ_4	22	31	27	4	4	***35***	4	4	***35***	4	4	27



(a) DMPO

(b) PA-DMMPT [10]

(c) Optimal approaches [7, 8]

Fig. 1. PTS schedule produced for the walk-through example task set of Table 1 by the assignment algorithms of (a) PA-DMMPT [10] and (b) optimal approaches such as TRAVERSE, PRUNED-TRAVERSE [8], and FAST-TRAVERSE. Note that task τ_4 in (a) and (b) misses its deadline while every task in (c) does not miss its deadline.

3. PREVIOUS ASSIGNMENT ALGORITHMS

There are largely two kinds of priority and preemption threshold assignment algorithms for PTS, which are heuristic ones and optimal ones. As discussed in Section 1.1, the best heuristic one is PA-DMMPT [10] and the best optimal one is PRUNED-TRAVERSE [8]. Therefore, we discuss further these two algorithms in this section. Both of PA-DMMPT and PRUNED-TRAVERSE first tentatively assign priorities to all tasks and then assign preemption thresholds to tasks using OPT-ASSIGN-THRESHOLD of [4]. When they assign priorities, both of them assign tasks priorities from the lowest priority 1 to the highest priority $|\Gamma|$.

PA-DMMPT assigns each priority to the task with the highest heuristic value among the remaining tasks in the unassigned task set. The heuristic value h_i of task τ_i is calculated as either the task blocking limit if $R_i \le D_i$ or $(D_i - R_i)$ otherwise. Here, all tasks in the unassigned task set except task τ_i are assigned deadline monotonic priorities that are higher than p_i. In addition to this, all tasks in the unassigned task set (including task τ_i) are assigned the maximum preemption thresholds. For the walk-through example task set in Table 1, PA-DMMPT assigns task priorities as follows.

- For priority 1, $h_1 = 9 - \varepsilon$, $h_2 = -18$, $h_3 = 0$, $h_4 = -17$. Thus, $p_1 = 1$.
- For priority 2, $h_2 = 13$, $h_3 = 13$, $h_4 = -17$.Thus, $p_2 = 2$ or $p_3 = 2$. We set $p_3 = 2$ since DMPO leads to the infeasible assignment as shown in Table 1.
- For priority 3, $h_2 = 18$, $h_4 = -12$. Thus, $p_2 = 3$.
- For priority 4, the only remaining task is τ_4. Thus, $p_4 = 4$.

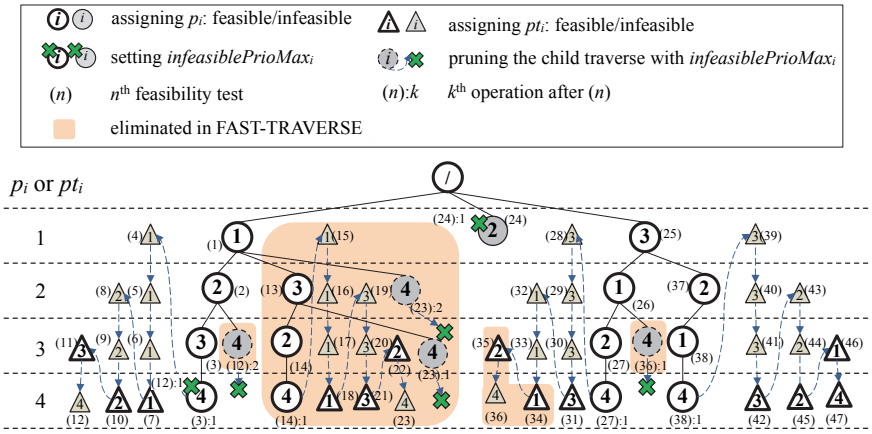
With this, the resultant priority ordering of PA-DMMPT is $PO_{PA} = \langle 1, 3, 2, 4 \rangle$. As shown in Table 1, priority ordering PO_{PA} makes task τ_4 miss its deadline since $(R_4 = 35) > (D_4 = 27)$. Fig. 1 (b) demonstrates such a deadline miss: the fifth instance of task τ_4 completes at time point 155 while its absolute deadline is $(5-1)\cdot T_4 + D_4 = 151$.

PRUNED-TRAVERSE traverses all possible priority orderings until it finds a feasible priority and preemption threshold assignment while pruning infeasible priority orderings. Specifically, it prunes any priority ordering that assigns priority *prio* to task τ_i in any of the following conditions.

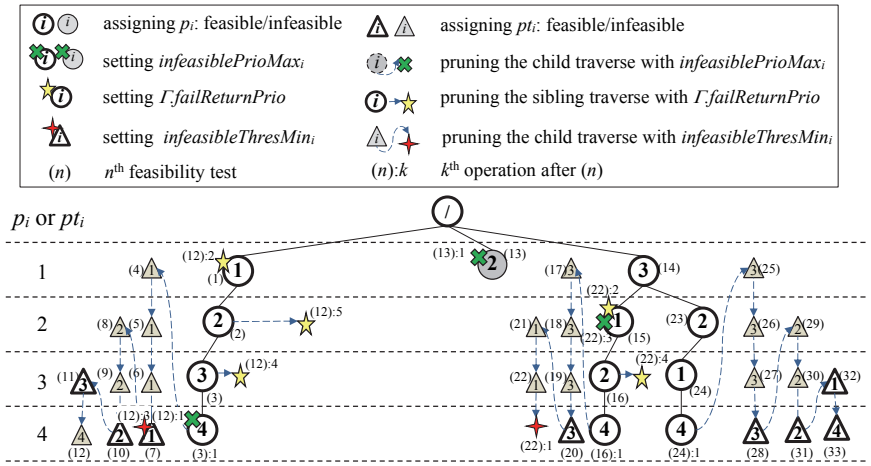
(Condition 1) Task τ_i is infeasible ($R_i > D_i$) with the highest preemption threshold.

(Condition 2) *prio* is less than $infeasiblePrioMax_i$, which is the maximum priority with which task τ_i has been infeasible.

For the walk-through example in Table 1, PRUNED-TRAVERSE assigns task priorities as shown in the priority and preemption threshold assignment tree in Fig. 2 (a). In the figure, each white/gray circle/triangle node represents a feasible/infeasible priority/preemption threshold assignment for a task, and its depth corresponds to the assigned priority/preemption threshold value. The green '*x*' mark annotation to a circle node represents the setting of the value of $infeasiblePrioMax_i$ to the priority level of the node. This annotation appears not only for gray (infeasible) circle nodes but also for the highest priority assignment circle node when its preemption threshold assignment is infeasible. Any green '*x*' mark connected with a dashed-lined gray circle node represents the pruning of the child traverse due to (Condition 2) with $infeasiblePrioMax_i$. Each path from the root to a leaf node corresponds to a possible task or priority ordering. Each number



(a) PRUNED-TRAVERSE



(b) FAST-TRAVERSE

Fig. 2. Priority and preemption threshold assignment trees for the walk-through example task set in Table 1.

with round braces "(n)" besides a node represents the nth feasibility test that is performed for the operation of the node, and "(n):k" besides a node or a mark denotes the kth operation after the nth feasibility test.

In Fig. 2 (a), four complete task orderings were generated: $TO_1 = \langle 1, 2, 3, 4\rangle$, $TO_2 = \langle 1, 3, 2, 4\rangle$, $TO_3 = \langle 3, 1, 2, 4\rangle$, and $TO_4 = \langle 3, 2, 1, 4\rangle$. The last task ordering $TO_4 = TO_{PR} = \langle 3, 2, 1, 4\rangle$ corresponds to the priority ordering $PO_{PR} = TO_{PR}^{-1} = \langle 3, 2, 1, 4\rangle$ for PRUNED-TRAVERSE of Table 1. The table shows that priority ordering PO_{PR} makes every task feasible, which is also demonstrated in Fig. 1 (c). As shown in Fig. 2 (a), whenever a complete task ordering is generated, preemption thresholds are assigned to the tasks. Specifically, for the task orderings TO_1-TO_4, four preemption threshold orderings are generated: $PTO_1 = \langle 4, 4, 3, (4)\rangle$, $PTO_2 = \langle 4, 3, 4, (4)\rangle$, $PTO_3 = \langle 4, 3, 4, (4)\rangle$, and $PTO_3 = \langle 3, 4, 4, 4\rangle$ where "(4)" represents an infeasible preemption threshold assignment. Note that the gray circles annotated with "(24)", "(12):2", "(23):1", "(23):2", and "(36):1"

pruned child traverses. Such pruning help in reducing the number of feasibility tests, which makes PRUNED-TRAVERSE outperform existing optimal scheduling attributes assignment algorithms. However, the nodes surrounded with round squares can be further eliminated in FAST-TRAVERSE, which will be shown in the next section.

4. ALGORITHM SPECIFICATION FOR FAST-TRAVERSE

FAST-TRAVERSE assigns tasks priorities from the lowest priority to the highest priority, which generates priority orderings until a feasible priority ordering is found. Once a priority ordering is found to be infeasible, it generates another priority ordering also from the lowest unassigned priority by a sibling traverse. Here a sibling traverse means assigning the same priority to the remaining tasks in the unassigned task set.

Our key idea is to prune sibling traverses if possible once we have found that a priority ordering is infeasible. For a given task set Γ that has been assigned a priority ordering, FAST-TRAVERSE finds the minimum priority Γ.*failReturnPrio* of the priority ordering that causes the task set Γ to be infeasible. Then, FAST-TRAVERSE prunes all sibling traverses that assign priorities larger than Γ.*failReturnPrio*. To find Γ.*failReturnPrio,* we also employ the notion of *effective* blocking task for priority level *prio*, which is the task that causes the task set to be infeasible by blocking a task with priority *prio*.

Along with sibling traverses, FAST-TRAVERSE also prunes preemption threshold assignments if possible using the notion of the minimum infeasible preemption threshold $infeasibleThresMin_i$ for task τ_i, which is the minimum preemption threshold of task τ_i that causes the task set to be infeasible. FAST-TRAVERSE sets this value whenever it finds an effective blocking task and prunes the preemption threshold assignment that assigns to task τ_i any preemption threshold not lesser than $infeasibleThresMin_i$.

Fig. 3 shows our proposed FAST-TRAVERSE in pseudo code. As shown in the figure, FAST-TRAVERSE invokes _FAST-TRAVERSE, which in turn invokes FAST-OPT-ASSIGN-THRESHOLD. FAST-TRAVERSE first sorts tasks using DMPO since DMPO finds feasible priority assignments in many cases (line 2). Then, it initializes parameters of each task (lines 3-6). Since priorities are in the range of [1, $|\Gamma|$], the initial maximum and minimum priority or preemption threshold values are set to 0 and $|\Gamma|$+1 (line 4), respectively. Like all existing priority assignment algorithms, FAST-TRAVERSE lets all priority unassigned tasks have the highest priority and preemption threshold (line 5). Then, it invokes _FAST-TRAVERSE.

_FAST-TRAVERSE assigns priorities *prio* from the lowest priority (1) to the highest priority ($|\Gamma|$) to each task τ_i in task set *UnAssigned* by recursively invoking itself with the priority *prio*+1 and the remaining unassigned task set *UnAssigned*−{τ_i} in line 19. Lines 9-20 and the last line 28 are the same as _PRUNED-TRAVERSE of except_ FAST-TRAVERSE invokes FAST-OPT-ASSIGN-THRESHOLD instead of RESOTRING-OPT-ASSIGN-THRESHOLD of [8]. Therefore, we only explain lines 21-27 in _FAST-TRAVERSE, which are added parts to _PRUNED-TRAVERSE of [8].

```
1   FAST-TRAVERSE (Γ: set of tasks)
2       Γ ← descendingSort(Γ,  D_i);
3       foreach (τ_i ∈ Γ)  {
```

```
4            infeasiblePrioMax_i ← 0;    infeasibleThresMin_i ← |Γ| + 1;
5            p_i ← |Γ|;      pt_i ← |Γ|;
6        } //end-foreach
7        return _FAST-TRAVERSE(1, Γ, Γ);
8    _FAST-TRAVERSE ( prio: priority,   UnAssigned: set of tasks,   Γ: set of tasks)
9        foreach ( τ_i ∈ UnAssigned) {
10           if ( prio ≤ infeasiblePrioMax_i )   continue;
11           p_i ← prio;    pt_i ← |Γ|;
12           if ( prio = |Γ| )   return FAST-OPT-ASSIGN-THRESHOLD(Γ);
13           if ( R_i > D_i ) {
14                p_i ← |Γ|;   // restore to the highest priority
15                if ( prio > infeasiblePrioMax_i )   infeasiblePrioMax_i ← prio;
16                continue;
17           } // end-if
18           pt_i ← prio;
19           if (_FAST-TRAVERSE(prio+1, UnAssigned −{ τ_i }, Γ) = success) return success;
20           foreach ( τ_j ∈ UnAssigned)   p_j ← |Γ|;   // restore to the highest priority
21           if ( prio > Γ.failReturnPrio )   return fail;   // prune the sibling traverse
22       } // end-foreach
23       τ_j ← ∀j, (pt_j ≥ prio > p_j) ∧ (C_j is the maximum); // τ_j is the effective blocking task.
24       if ( τ_j exists)   {
25           Γ.failReturnPrio ← p_j;   // from Theorem 1
26       infeasibleThresMin_j ← pt_j;   // from Theorem 2
27       } else Γ.failReturnPrio ← 0; // Γ is not feasible with any assignment.
28       return fail;
29   FAST-OPT-ASSIGN-THRESHOLD (Γ: set of tasks)
30       SortedTaskSet ← ascendingSort(Γ, p_i);
31       foreach ( τ_i ∈ SortedTaskSet)   {
32           pt_i ← p_i;
33           while (R_i > D_i )) {
34               pt_i ← pt_i + 1;
35               if ((pt_i >|Γ|) or (pt_i ≥ infeasibleThresMin_i))) {
36                   if (p_i > infeasiblePrioMax_i )   infeasiblePrioMax_i ← p_i;
37                   Γ.failReturnPrio ← p_i;
38                   if (p_i ==|Γ|) {
39                       τ_j ← ∀j, (pt_j ≥ |Γ| > p_j) ∧ (C_j is the maximum);
40                       Γ.failReturnPrio ← p_j; // from Corollary 1
41                       infeasibleThresMin_j = pt_j; // from Theorem 2
42                   } // end-if
43                   return fail;
44               } // end-if } // end-while   } //end-foreach
45       return success;
```

Fig. 3. Pseudo code for the proposed FAST-TRAVERSE algorithm.

Table 2. An infeasible task set under PTS even with the optimal assignment.

	C_i	T_i	D_i
τ_1	4	640	400
τ_2	11	160	100
τ_3	23	100	90
τ_4	2	3	3

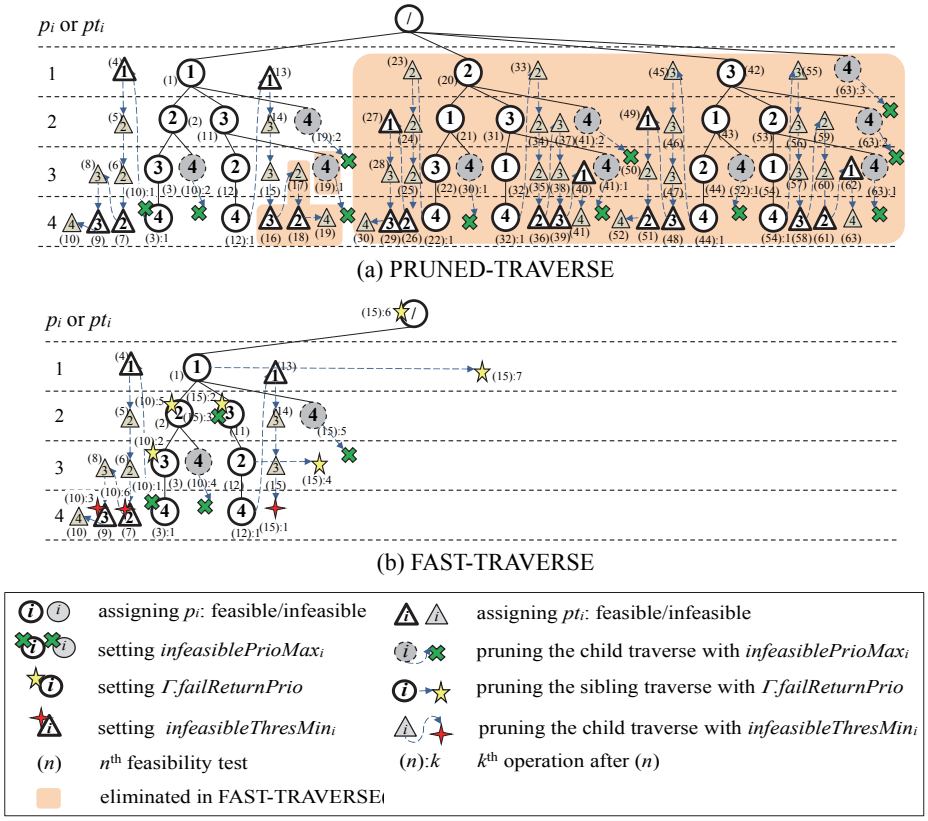


Fig. 4. Priority and preemption threshold assignment trees for the infeasible example task set in Table 2.

First, line 21 lets the algorithm return if $prio > \Gamma.failReturnPrio$, which prunes the sibling traverse for assigning priority $prio$ for the remaining tasks in *UnAssigned* in line 9. Lines 23-28 are executed when all tasks in *UnAssigned* are pruned in assigning priority $prio$. If the effective blocking task τ_j for the priority level $prio$ exists (line 24), $\Gamma.failReturnPrio$ and $infeasibleThresMin_j$ are set to p_j (line 25) *and* pt_j (line 26), respectively, from the following Theorems 1 and 2. Otherwise, $\Gamma.failReturnPrio$ is set to 0 (line 27) since Γ is not feasible with any assignment.

Lemma 1: (From Theorem 2 of [8]) Under PTS, if preemption threshold pt_i of task τ_i is fixed, then its worst-case response time R_i does not decrease when its priority p_i is lowered.

Theorem 1: Under any scheduling attributes assignment algorithm that assigns distinctive priorities to tasks for PTS, if all tasks are pruned while assigning priority $prio$ and there exists an effective blocking task τ_j, then $\Gamma.failReturnPrio$ is the priority of p_j.

Proof: Let a pruned task for assigning priority $prio$ be τ_i. Since all tasks have been

pruned while assigning priority *prio*, the next priority assignment iteration would try to assign a lower priority to task τ_i. From Lemma 1, R_i does not decrease when its priority is lowered (since its interference time always exceeds its blocking time from the proof of Lemma 1). Accordingly, the only way to decrease R_i is to reduce blocking time B_i, which is C_j as long as there exists τ_j. Therefore, we don't need to traverse sibling priority assignment that assigns task τ_i the lower priority than *prio* until we eliminate the previous priority assignment to task τ_j. Consequently, Γ.*failReturnPrio* is p_j. ❑

Theorem 2: Under any scheduling attributes assignment algorithm that assigns distinctive priorities to tasks for PTS, if all tasks are pruned while assigning priority *prio* and there exists effective blocking task τ_j, then *infeasibleThresMin*$_j$ is pt_j.

***Proof*:** This theorem can be proved in a similar way to the proof of Theorem 1. The existence of task τ_j and pruning of all tasks while assigning priority *prio* infers that the pruned tasks cannot be feasible without reducing their blocking times. Since their blocking times were caused by the preemption threshold assignment to task τ_j, *infeasibleThresMin*$_j$ is pt_j. ❑

FAST-OPT-ASSIGN-THRESHOLD is the same as the optimal preemption threshold assignment algorithm of [4] except the additional second condition of line 35 and additional operations of lines 37-42. First, the second condition of line 35, $pt_i \geq$ *infeasibleThresMin*$_i$, is for additionally pruning the preemption threshold assignment when the currently assigning preemption threshold for task τ_i is not lesser than *infeasibleThresMin*$_i$. Line 36 is from _PRUNED-TRAVERSE of [8] and updates *infeasiblePrioMax*$_i$ of task τ_i since priority p_i is also infeasible priority of task τ_i. In line 37, Γ.*failReturnPrio* is set to p_i since the priority ordering is infeasible due to the priority assignment to task τ_i. If task τ_i is the highest priority task (line 38), the effective blocking task τ_j always exists. In this case, Γ.*failReturnPrio* and *infeasibleThresMin*$_j$ are set to p_j (line 40) *and* pt_j (line 41), respectively from the following Corollary 1 and the above Theorem 2.

Corollary 1: Under any scheduling attributes assignment algorithm that assigns distinctive priorities to tasks for PTS, if the highest priority task τ_h is infeasible, then Γ.*failReturnPrio* is the priority of the effective blocking task.
Proof. If the highest priority task τ_h is infeasible, there is no remaining task to be assigned priority, which means all tasks are pruned while assigning priority $|\Gamma_i|$ and there is always exists an effective blocking task. Therefore, this corollary follows from Theorem 1. ❑

For the walk-through example task set in Table 1, FAST-TRAVERSE assigns task priorities as shown in Fig. 2 (b). The figure uses additional graphical notations to those of Fig. 2 (a). Specifically, the yellow star annotation to a white circle node represents setting the value of Γ.*failReturnPrio* to the priority level of the node. Any yellow star mark connected with a white circle node represents pruning the sibling traverse with Γ.*failReturnPrio*. The red cross mark annotation to a white triangle node represents setting the value of *infeasibleThresMin*$_i$ to the priority level of the node. Any red cross mark connected with a gray triangle node represents pruning the preemption threshold assignment with *infeasibleThresMin*$_i$.

Figs. 2 (a) and (b) are the same from the node of "(1)" to the 'x' mark annotation of "(12):1". However, in Fig. 2 (b), the yellow star mark "(12):2" identifies the effective blocking as τ_1 (line 39 in Fig. 3) and sets $\Gamma.failReturnPrio = p_1 = 1$ (line 40 in Fig. 3). Then, the red cross mark annotation of "(12):3" sets $infeasibleThresMin_1 = pt_1 = 4$ (line 41 in Fig. 3). After this, the yellow star marks of "(12):4" and "(12):5" eliminate sibling traverses since priority levels of 3 and 2 are larger than $\Gamma.failReturnPrio$ (= 1) (line 21 in Fig. 3). Moreover, the red cross mark annotation of "(22):1" eliminates the nodes from "(34)" to "(36)" in Fig. 2 (a) since priority level 4 is the same as $infeasibleThresMin_1$ (= 4) (lines 35 and 43 in Fig. 3). In such a way, nodes surrounded by four round squares in Fig. 2 (a) were eliminated in Fig. 2 (b).

As another example, we introduce an infeasible task set under PTS even with any optimal assignment algorithm in Table 2. Note that task indicess of the task set are in the deadline monotonic decreasing order (DMPO). Fig. 4 shows priority and preemption threshold assignment trees for this task set. In Fig. 4, FAST-TRAVERSE finds all tasks are pruned while assigning specific priority twice. The first is after "(10):4" and the other is after "(15):5" while assigning priorities 3 and 2, respectively. In the first case where assigning priority 3, the yellow star mark "(10):5" identifies the effective blocking as τ_2 and sets $\Gamma.failReturnPrio = p_2 = 2$ (line 25 in Fig. 3). Due to this, the yellow star marks of "(15):4" eliminates the sibling traverse since priority level 3 is larger than $\Gamma.failReturnPrio$ (= 2) (line 21 in Fig. 3).

In the other case where assigning priority 2 (after "(15):5"), there does not exist the effective blocking task. Therefore, FAST-TRAVERSE sets $\Gamma.failReturnPrio = 0$ (line 27 in Fig 3), as shown with the yellow start mark annotation of "(15):6". Note that this setting of $\Gamma.failReturnPrio = 0$ enables pruning of the sibling traverse as shown with the yellow star mark annotation of "(15):7", which greatly helps in eliminating a large number of feasibility tests in Fig. 4 (a).

4.1 Complexity

The complexity of PRUNED-TRAVERSE has been shown to be $O(E \cdot n! \cdot n^2)$ [8] where $n=|\Gamma|$ and E is the non-polynomial [3] complexity of the feasibility test (calculating R_i from Eqs. (1)-(6)). $O(E \cdot n! \cdot n^2)$ is derived from $A + B \cdot C$ where A is the complexity of priority assignment, B is the number of priority orderings, and C is the complexity of preemption threshold assignment. Each value is derived as follows.

- The complexity of priority assignment (A):

$$O(E \cdot n \cdot \{1 + (n-1) \cdot \{1 + (n-2) \cdot \{1 + \dots\}\}\})$$
$$= O(E \cdot \{n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \dots + n!\})$$
$$= O(E \cdot n!)$$

- The number of priority orderings (B): $n!$
- The complexity of preemption threshold assignment (C):

$$O(E \cdot \{n + (n-1) + \dots + 1\})$$
$$= O(E \cdot n \cdot (n-1)/2)$$
$$= O(E \cdot n^2)$$

The complexity of FAST-TRAVERSE is derived in a similar way. Let the values of A, B, and C for FAST-TRAVERSE be A', B', and C' and then the complexity of FAST-TRAVERSE is $A' + B' \cdot C'$. Once the first complete priority ordering and the preemption threshold assignment does not succeed (returns fail), FAST-TRAVERSE finds the effective blocking task τ_j. Then, $\Gamma.failReturnPrio$ and $infeasibleThresMin_j$ is set. In addition to this, for any task τ_k that has been found infeasible, $infeasiblePrioMax_k$ is set. With this, $\Gamma.failReturnPrio$ and $infeasiblePrioMax_k$ affects A' and B' while $infeasibleThresMin_j$ affects C'. Among these, the effects of $\Gamma.failReturnPrio$ are dominant and thus we consider its effect as follows.

If $\Gamma.failReturnPrio$ is set to 0, which has been demonstrated as the yellow start mark annotation of "(15):6" in Fig. 4 (b), any further traverse is stopped, which means the further traverse takes time of 0. If $\Gamma.failReturnPrio$ is set to 1, which has been demonstrated as the yellow start mark annotation of "(12):2" in Fig. 2 (b), all sibling traverses of assigning priorities $2 - n$ are pruned. The value of $\Gamma.failReturnPrio$ is set from 0 to $n - 1$ and the probability of each value can be regarded to be in a uniform distribution, which means each value has the possibility of $1/n$. Then, the possibility of non-pruning is $1-1/n$. Then, the probabilistic number of assigning priority 1 is $1+(1 - 1/n)\cdot\{1 + (1 - 1/n)\cdot\{1 + (1 - 1/n)\cdot\{\ldots\}\} = \sum_{k=1}^{n}(1-1/n)^{k-1} = n\cdot(1 - ((n - 1)/n)^n)$. In such a way, for each assignment of priority i, the next probabilistic number of assigning priority $i+1$ is $\sum_{k=1}^{n-i}(1-1/(n-i))^{k-1} = (n - i)\cdot(1 - ((n - i - 1)/(n - i)^{n-i})$. With this, A', B', and C' are calculated as follows.

- The complexity of priority assignment (A'):

$$
\begin{aligned}
&\mathrm{O}\left(E\cdot\left\{n\cdot\left(1-\left(\frac{n-1}{n}\right)^{n}\right)\cdot\left\{1+(n-1)\cdot\left(1-\left(\frac{n-2}{n-1}\right)^{n-1}\right)\cdot\{1+\ldots\}\right\}\right\}\right)\\
&=\mathrm{O}\left(E\cdot\left\{n\cdot\left(1-\left(\frac{n-1}{n}\right)^{n}\right)+n\cdot\left(1-\left(\frac{n-1}{n}\right)^{n}\right)\cdot(n-1)\cdot\left(1-\left(\frac{n-2}{n-1}\right)^{n-1}\right)+\ldots\right\}\right)\\
&=\mathrm{O}\left(E\cdot n\cdot\left(1-\left(\frac{n-1}{n}\right)^{n}\right)\cdot(n-1)\cdot\left(1-\left(\frac{n-2}{n-1}\right)^{n-1}\right)\cdot\ldots\cdot 2\cdot\left(1-\left(\frac{1}{2}\right)^{2}\right)\cdot 1\cdot\left(1-\left(\frac{0}{1}\right)^{1}\right)\right)\\
&=\mathrm{O}\left(E\cdot n!\cdot\prod_{k=0}^{n-1}\left(1-\left(\frac{n-k-1}{n-k}\right)^{n-k}\right)\right)
\end{aligned}
$$

- The number of priority orderings (B'): $n!\cdot\prod_{k=0}^{n-1}(1-(\frac{n-k-1}{n-k})^{n-k})$. While $infeasiblePrio\ Max_k$ further reduces the number of priority orderings, its probability is unpredictable and thus we ignore its effect for the complexity computation.
- The complexity of preemption threshold assignment (C'): the same as $C = \mathrm{O}(E\cdot n^2)$. While $infeasibleThresMin_j$ ensures that C' is less than C, its probability is unpredictable and thus we regard C' as C.

Therefore, the complexity of FAST-TRAVERSE, $A' + B'\cdot C'$, is as follows.

$$
\mathrm{O}\left(E\cdot n!\cdot n^2\cdot\prod_{k=0}^{n-1}\left(1-\left(\frac{n-k-1}{n-k}\right)^{n-k}\right)\right)
$$

Since $1-(\frac{n-k-1}{n-k})^{n-k}$ is less than 1, such multiplications to the complexity of PRUNED-TRAVERSE reduces much the computation time of PRUNED-TRAVERSE. For example, when n is 5, 10, and 20, the values of $\prod_{k=0}^{n-1}(1-(\frac{n-k-1}{n-k})^{n-k})$ are about 0.24, 0.03, and 0.0004, respectively. Such a reduction will be empirically shown in the next section.

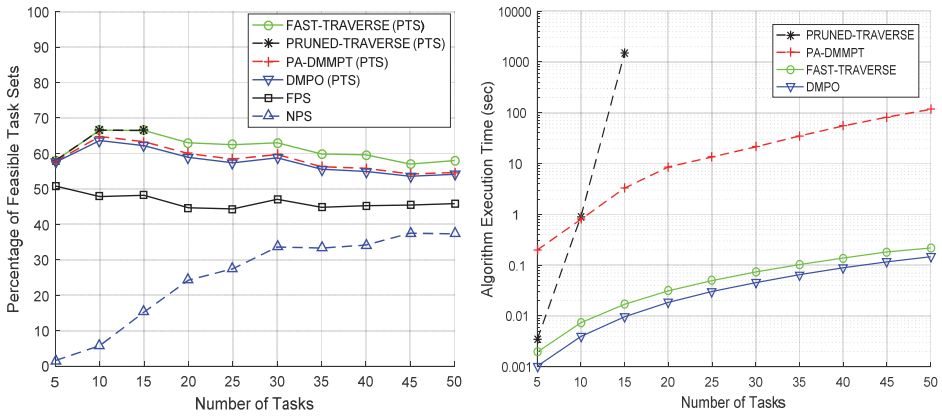
5. EMPIRICAL PERFORMANCE EVALUATION

This section empirically evaluates the performance of FAST-TRAVERSE by comparing it with DMPO, PA-DMMPT, and PRUNED-TRAVERSE, which were discussed in Sections 1.1 and 3. The performance metrics are 1) the percentage of feasible task sets and 2) the actual run time for the algorithm execution. The experiments were conducted with Matlab R2016a on an Intel Core i7-4790, 3.60 GHz system with 16 GB of RAM. To ensure that the feasibility of a given task set is greatly dependent on a proper scheduling attributes assignment algorithm, we need to prepare highly demanding workloads. Therefore, we set the total utilization of each task set to $U = 0.9$. We have generated each task set in the similar manner as in [9]. Specifically, for each task τ_i, we generated C_i as a random integer that is uniformly distributed in the interval [100, 500] and the utilization of each task u_i using UUniFast [19] algorithm. Accordingly, we derived $T_i = C_i/u_i$ while D_i as a random integer that is uniformly distributed in the interval $[C_i + 0.5 \cdot (T_i - C_i), T_i]$. We varied the number of tasks $|\Gamma|$ from five to fifty. For each configuration, we generated 2,000 task sets. For the feasibility test, we set the value of ε in Eq. (6) to 10^{-6} (ε was 1 in [9]).

Fig. 5 (a) shows the percentage of feasible task sets for varying number of tasks. Due to the run-time performance problem, the results of PRUNED-TRAERSE were only available when the number of tasks is not larger than 15. As shown in the figure, the percentages of feasible task sets are the same for PRUNED-TRAVERSE and FAST-TRAVERSE when the results of PRUNED-TRAVERSE are available. Fig. 5 clearly shows that FAST-TRAVERSE always has the larger percentage of feasible task sets than PA-DMMPT and DMPO. Specifically when $|\Gamma| = 25$, FAST-TRAVERSE could make 4.1% and 5.1% more task sets feasible than PA-DMMPT and DMPO, respectively (62.5% for FAST-TRAVERSE, 58.4% for PA-DMMPT, and 57.4% for DMPO).

Fig. 5 (b) shows the average algorithm run time results for varying number of tasks. Y axis is in a log scale to better show the distributions of result values. Fig. 5 (b) clearly shows that FAST-TRAVERSE is much faster than PRUNED-TRAVERSE and PA-DMMPT. When $|\Gamma| = 15$, the average algorithm run times with FAST-TRAVERSE were reduced by 8,802,627% and 19,185% compared to PRUNED-TRAVERSE and PA-DMMPT, respectively (0.017 sec for FAST-TRAVERSE, 1,496.5 sec for PRUNED-TRAVERSE, and 3.278 sec for PA-DMMPT). Fig. 5 (b) also shows that FAST-TRAVERSE even takes similar algorithm execution time to DMPO, which is one of the simplest and widely used priority assignment algorithms for PTS as we discussed in Section 1.1. More importantly, the average algorithm run time differences between FAST-TRAVERSE and DMPO change little as the number of tasks $|\Gamma|$ increases. Even when $|\Gamma| = 50$, the average algorithm run time of FAST-TRAVERSE was 0.2178 sec, which is only 49% larger than that of DMPO (0.1462 sec), while it always produces an optimal priority

assignment result. With this, FAST-TRAVERSE can be used as an on-line priority and preemption threshold assignment algorithm for PTS, which is essential for on-line admission control for newly arrived tasks.



(a) The ratio of feasible tasks. (b) Algorithm execution time.

Fig. 5. Performance evaluation with varying number of tasks ($U = 0.9$).

Table 3. Comparison of the average numbers of response time tests (iteration counts) over the varying number of tasks.

Number of tasks	5	10	15
PRUNED-TRAVERSE	22.1	2614.4	2340017.8
FAST-TRAVERSE	11.7	24.5	34.6
Ratio of the number of the eliminated tests of PRUNED-TRAVERSE in FAST-TRAVERSE	47.1%	99.06%	99.9985%

Table 3 compares the average number of response time tests (iteration counts) by using Eqs. (1)-(6) for PRUNED-TRAVERSE and FAST-TRAVERSE over varying number of tasks. The last row of Table 3 shows the ratio of the number of the eliminated tests of PRUNED-TRAVERSE in FAST-TRAVERSE. As shown in the table, the more the number of the tasks is, the reduction ratio becomes dramatically larger.

6. CONCLUSIONS

We proposed an optimal priority and preemption threshold assignment algorithm for PTS, which we named FAST-TRAVERSE. The proposed algorithm is optimal: it always finds a feasible priority and preemption threshold assignment if one exists. FAST-TRAVERSE prunes sibling traverses and preemption threshold assignments using the notions of the effective blocking task. The empirical evaluation results clearly showed that FAST-TRAVERSE always makes more task sets feasible than any other non-optimal priority assignment algorithm for PTS. The empirical results also showed that FAST-TRAVERSE is applicable for a large number of tasks, which was impossible with

previous existing optimal algorithms. We also showed that FAST-TRAVERSE is much faster than the best known heuristic algorithm, which makes it applicable as an on-line admission control algorithm.

REFERENCES

1. M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proceedings of IEEE Real-Time Systems Symposium*, 2000, pp. 25-34,
2. S. Kim, "Dual ceiling protocol for real-time synchronization under preemption threshold scheduling," *Journal of Computer and System Sciences*, Vol. 76, 2010, pp. 741-750.
3. N. C. Audsley, "On priority assignment in fixed priority scheduling," *Information Processing Letters*, Vol. 79, 2001, pp. 39-44.
4. Y. Wang and M. Saksena, "Scheduling fixed priority tasks with preemption threshold," in *Proceedings of IEEE Real-Time Computing Systems and Applications Symposium*, 1999, pp. 328-335.
5. K. Bletsas and N. Audsley, "Optimal priority assignment in the presence of blocking," *Information Processing Letters*, Vol. 99, 2006, pp. 83-86.
6. A. Zuhily and A. Burns, "Optimal (D−J)-monotonic priority assignment," *Information Processing Letters*, Vol. 103, 2007, pp. 247-250.
7. S. Kim, "Synthesizing multithreaded code from real-time object-oriented models via schedulability-aware thread derivation," *IEEE Transactions on Software Engineering*, Vol. 40, 2014, pp. 413-426.
8. S. Kim, "Assigning priorities for fixed priority preemption threshold scheduling," *The Scientific World Journal*, Vol. 2015, 2015, pp. 1-14.
9. G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems, a survey," *IEEE Transactions on Industrial Informatics*, Vol. 9, 2013, pp. 3-15.
10. 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, 2014, pp. 149:1-149:25.
11. N. Min-Allaha, S. U. Khanb, X. Wangc, and A. Y. Zomayad, "Lowest priority first based feasibility analysis of real-time systems," *Journal of Parallel and Distributed Computing*, Vol. 73, 2013, pp. 1066-1075.
12. R. J. Bril, S. Altmeyer, V. Heuvel, M. M. H. P. van den Heuvel, and R. I. Davis, "Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds," in *Proceedings of Real-Time Systems Symposium*, 2014, pp. 161-172.
13. J. Chen, A. Harji, and P. Buhr, "Solution space for fixed-priority with preemption threshold," in *Proceedings of IEEE Real Time and Embedded Technology and Applications Symposium*, 2005, pp. 385-394.
14. U. Keskin, R. J. Bril, and J. J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *Proceedings of IEEE Conference on Emerging Technologies and Factory Automation*, 2010, pp. 1-4.

15. J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proceedings of IEEE Real-Time Systems Symposium*, 2002, pp. 315-326.
16. J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of IEEE Real-Time Systems Symposium*, 1990, pp. 201-209.
17. E. Bini, T. H. C. Nguyen, P. Richard, and S. K. Baruah, "Response-time bound in fixed-priority scheduling with arbitrary deadlines," *IEEE Transactions on Computers*, Vol. 58, 2009, pp. 279-286.
18. C. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment," *Journal of the ACM*, Vol. 20, 1973, pp. 46-61.
19. E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, Vol. 30, 2005, pp. 129-154.

Saehwa Kim (金世和) received her B.S., M.S., and Ph.D. degrees in Electrical and Computer Science Engineering from Seoul National University, Seoul, Korea, in 1997, 2000, and 2006, respectively. She is currently an Associate Professor at the Department of Information Communications Engineering, Hankuk University of Foreign Studies, Korea. Her research interest is in software engineering and embedded software platforms that are specialized for specific industrial domains such as automotive vehicles, intelligent robots, and software defined radios.