

# Preemption Threshold Assignment to Improve Schedulability under Memory Constraints

Thilanka Thilakasiri, Matthias Becker  
*KTH Royal Institute of Technology, Stockholm, Sweden*  
 {thilanka, mabecker}@kth.se

**Abstract**—In this paper, we propose a novel preemption threshold assignment algorithm that considers both the memory limitation and schedulability, thereby improving both aspects as opposed to the state-of-the-art algorithms that only consider one of the two aspects. In addition, the proposed algorithm explores only a fraction of preemption threshold configurations in a shorter time compared to the state-of-the-art.

**Index Terms**—limited preemption, preemption thresholds

## I. INTRODUCTION

Non-preemptive (NP) scheduling minimizes the memory usage by running all the tasks from the same shared stack (a single task at a time); however blocking in non-preemptive scheduling reduces schedulability. While fully-preemptive (FP) scheduling reduces blocking and improves schedulability, it requires a larger memory to accommodate the stack of each task. Limited preemption is a favorable middle path to reduce memory requirements by limiting preemptions while maintaining schedulability [1], [2], e.g., preemption points [3], deferred preemption [4], and preemption thresholds [5], [6]. This paper focuses on preemption thresholds (PT). PTs temporarily increase the priority of executing tasks, thereby limiting the number of tasks that can preempt them [5]–[9]. PTs have also been commonly utilized in the literature to reduce memory/stack usage, especially for stack sharing in RTOS [5], [7], [10]–[13]. Using an effective preemption threshold assignment algorithm that addresses all system objectives, e.g., timing and memory aspects, is crucial. The maximum preemption threshold assignment algorithm (MPTAA) [11] assigns maximum possible preemption thresholds to tasks, aiming to improve memory/stack usage. MPTAA is based on the maximum preemption threshold algorithm proposed by Wang and Saksena [5] and has been proven to find the largest preemption threshold assignment than any other feasible preemption threshold assignment [14]. MPTAA starts with a task set that is already schedulable and increases the PTs while maintaining the schedulability it initially had improving the memory requirement. Thus, independent of the initial configuration used (NP or FP), MPTAA is suboptimal in the schedulability aspect as neither NP nor FP dominates the other [6], [15]. Wang and Saksena have proposed another algorithm, which we call the preemption threshold assignment algorithm for schedulability (PTAS), that has an FP initial configuration and increases the PTs, aiming to find a schedulable PT configuration (see Figure 2 in [6]). However, improving the memory aspect is not an objective of PTAS, i.e., it returns the lowest PT that makes the task set schedulable.

Thus, existing algorithms for preemption threshold assignment either focus on improving memory usage while maintaining the initial schedulability or on improving schedulability without considering memory limitations. We propose the Non-Preemptive First Threshold Assignment Algorithm (NPFTA), which aims to improve schedulability as well as memory usage.

## II. SYSTEM MODEL

We consider an application with  $n$  independent sporadic tasks  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$  scheduled by preemptive fixed-priority scheduling on a single-core platform. Each task is represented using the tuple  $(T_i, D_i, S_i, P_i, \theta_i, C_i)$ , denoting the task's minimum inter-arrival time, relative deadline ( $D_i \leq T_i$ ), the maximum stack usage, the fixed nominal priority, tasks's preemption threshold ( $P_i \leq \theta_i$ ), and the task's worst-case execution time (WCET). Higher values indicate higher priority. As we consider preemption thresholds (PT), the priority of a task  $\tau_i$  is raised to its PT,  $\theta_i$ , at the start of its execution. The priority is restored to its nominal value after the execution finishes.

## III. THE PROPOSED NPFTA ALGORITHM

NPFTA has an NP initial configuration, i.e., the PTs of all tasks are configured to the maximum priority in the task set. Starting with an NP configuration and reducing the PTs if needed can improve schedulability, memory usage, search space, and time to find a solution when assigning PT due to the following reasons: (1) NP scheduling results in minimum memory/stack usage. (2) Task sets already schedulable under NP scheduling do not need further modification. (3) Task sets that are schedulable under other configurations than the initial configuration can be found, in contrast to MPTAA.

---

**Algorithm 1** Non-Preemptive First Threshold Assignment Algorithm (NPFTA)

---

```

1: Input: TaskSet
2: TaskSet  $\leftarrow$  assignNPPT() ▷ initial non-preemptive PTs
3: if sched == True then return Success
4: else
5:   missedTasks, otherTasks  $\leftarrow$  sortTasks()
6:   if len(otherTasks) == 0 then return Fail
7:   else
8:     sched  $\leftarrow$  Algorithm2(missedTasks, otherTasks)
9:     if sched == False then return Fail
10:    else return Success
11:    end if
12:  end if
13: end if

```

---

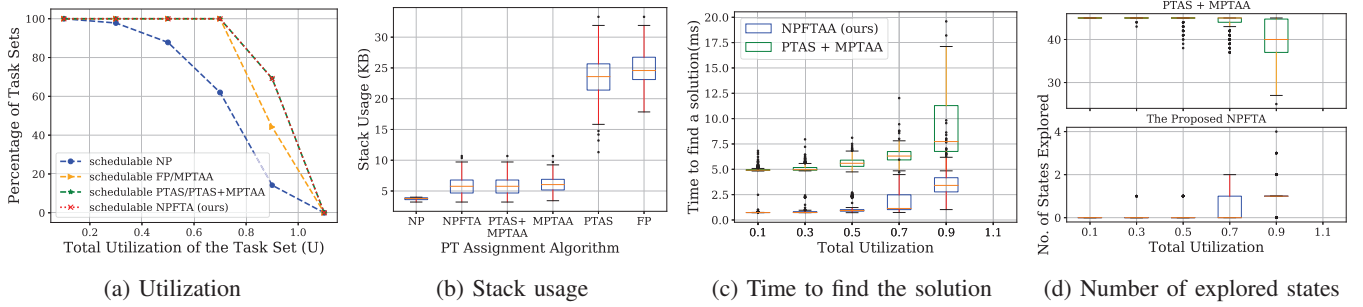


Fig. 1: (a)-Percentages of task sets that are schedulable. (b)-Stack Usages calculated using the analysis in [13] at  $U=0.9$ . (c)-Time to find a solution for the same experiment as (a). (d)-No. of states explored for the same experiment as (a).

### Algorithm 2 Recursive PT adjustment for schedulability

```

1: Input: missedTasks, otherTasks
2: otherTasks.sort()  $\triangleright$  sort according to the WCET
3:  $LPrio \leftarrow getLowestPriority(missedTasks)$   $\triangleright$  find the lowest priority
   among the missed tasks
4:  $nextTask \leftarrow getNextTask(otherTasks)$ 
5: while  $nextTask$  found and  $sched == False$  do
6:    $nextTask.setPT(LPrio - 1)$ 
7:    $sched \leftarrow WCRT\_Analysis(TaskSet)$ 
8:    $Algorithm2(missedTasks', otherTasks')$   $\triangleright$  recursive
9: end while
10: return  $sched$ 

```

NPFTA takes the task set as the input (see Algorithm 1). It returns whether a schedulable PT configuration is found for a task set, i.e., *Success*, and *Fail* otherwise. First the task set's schedulability is evaluated at the initial NP configuration using the WCRT in [8]. In the case where the task set meets all deadlines under the NP configuration, the algorithm looks no further and returns *Success*, otherwise it looks further into configuring PTs to make the tasks schedulable. A list of tasks that missed their deadline (*missedTasks*) and the rest of the tasks that met their deadlines (*otherTasks*) are identified. When not all tasks miss their deadlines, we reduce the PTs of suitable tasks (that met their deadline) to reduce blocking time on those that missed their deadlines, aiming to improve schedulability (Line 8 in Algorithm 1). This is done recursively (see Algorithm 2). Often, it is higher-priority tasks that miss the deadline in NP scheduling due to blocking from lower-priority tasks with larger WCETs. Thus, it is effective to reduce the PTs of lower-priority tasks that have longer WCETs compared to the missed tasks, aiming to reduce blocking. Another reason for this is that tasks with longer execution times often have longer periods and can often afford to be preempted without affecting their schedulability. In order to make the task set schedulable, even the task that has the lowest nominal priority ( $LPrio$ ) among the tasks that missed the deadlines should meet its deadline. Thus, the PT of a task that has a lower nominal priority must be decremented below  $LPrio$ .  $getNextTask()$  finds the next task to reduce the PT of ( $nextTask$ ). Which is the task with the longest WCET that has a higher or equal PT than  $LPrio$  and a nominal priority lower than  $LPrio$ , i.e.,  $\theta_{nextTask} \geq LPrio > P_{nextTask}$ . In other words,  $getNextTask()$  finds the lower-priority task that the missed

tasks couldn't preempt, which is potentially causing the most blocking. In this process of finding the *nextTask*, we first look among the tasks that have a non-preemptive PT configuration (PT at the highest priority level). If no task among them satisfies the conditions then we look at the tasks whose PTs have already been adjusted. Once the *nextTask* is found, we reduce its PT to  $LPrio - 1$  (see Line 6 in Algorithm 2) so that preemption could help all missed tasks to meet their deadlines by reducing blocking. The schedulability is evaluated after each PT change. If there are tasks that miss the deadline still, we recursively call Algorithm 2 until the tasks are schedulable or until we identify that the task set is not schedulable, i.e., no more tasks found to adjust the PT of (Line 8 in Algorithm 2).

## IV. RESULTS AND DISCUSSION

Each evaluated task set consists of 10 tasks, and their minimum inter-arrival times follow a log-uniform distribution in the range [100, 1000]. The generated stack sizes follow a uniform distribution in the range [1KB, 4KB]. Rate-monotonic priorities are used. We evaluate our NPFTA algorithm compared to MPTAA (with a FP initial configuration as commonly used in literature [10], [11], [16], [17]), PTAS and PTAS+MPTAA (using the result of PTAS as the input to MPTAA, which is not evaluated in literature to the best of our knowledge) on the schedulability, stack usage, explored preemption threshold state space, and time to find a solution (see Figure 1). Each plotted point represents 500 task sets. NP scheduling provides the best memory usage but the lowest schedulability. FP scheduling has better schedulability compared to NP scheduling, but has the worst memory usage. PTAS gives the same schedulability as NPFTA but has a high memory usage. MPTAA has better memory usage, but the schedulability can be improved at higher utilizations. The proposed NPFTA and the PTAS+MPTAA offer the best schedulability while also minimizing memory usage. Among NPFTA and PTAS+MPTAA, the proposed NPFTA explores only a fraction of the PT configurations that PTAS+MPTAA explores in a shorter time, making it suitable for larger systems and for design space exploration as well. To conclude, the proposed NPFTA algorithm improves both schedulability and memory efficiency, while also reducing the number of explored states and the time to compute a solution; objectives that have, to the best of our knowledge, not been jointly addressed by any existing approach.

## REFERENCES

- [1] J. Lee and K. G. Shin, "Controlling preemption for better schedulability in multi-core systems," in *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE, 2012, pp. 29–38.
- [2] G. Buttazzo, *Limited Preemptive Scheduling*. Springer Nature Switzerland, 2024, pp. 229–261.
- [3] A. Burns, *Preemptive priority based scheduling: An appropriate engineering approach*. Citeseer, 1993.
- [4] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *17th euromicro conference on real-time systems (ECRTS'05)*. IEEE, 2005, pp. 137–144.
- [5] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proceedings 21st IEEE real-time systems symposium*. IEEE, 2000, pp. 25–34.
- [6] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999, pp. 328–335.
- [7] R. Davis, N. Merriam, and N. Tracey, "How embedded applications using an rtos can stay within on-chip memory limits," in *12th EuroMicro Conference on Real-Time Systems*, 2000, pp. 71–77.
- [8] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002*. IEEE, 2002, pp. 315–326.
- [9] G. Yao and G. Buttazzo, "Reducing stack with intra-task threshold priorities in real-time systems," in *Proceedings of the tenth ACM international conference on Embedded software*, 2010, pp. 109–118.
- [10] C. Wang, C. Dong, H. Zeng, and Z. Gu, "Minimizing stack memory for hard real-time applications on multicore platforms with partitioned fixed-priority or edf scheduling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, May 2016. [Online]. Available: <https://doi.org/10.1145/2846096>
- [11] R. Ghattas and A. G. Dean, "Preemption threshold scheduling: Stack optimality, enhancements and analysis," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, 2007, pp. 147–157.
- [12] C. Dong and H. Zeng, "Minimizing stack memory for hard real-time applications on multicore platforms," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.
- [13] S. Altmeyer, R. J. Bril, and P. Gai, "Empress: an efficient and effective method for predictable stack sharing," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018, pp. 92–100.
- [14] J. Chen, A. Harji, and P. Buhr, "Solution space for fixed-priority with preemption threshold," in *11th IEEE Real Time and Embedded Technology and Applications Symposium*. IEEE, 2005, pp. 385–394.
- [15] 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, 2012.
- [16] C. Wang, C. Dong, H. Zeng, and Z. Gu, "Minimizing stack memory for hard real-time applications on multicore platforms with partitioned fixed-priority or edf scheduling," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 3, pp. 1–25, 2016.
- [17] C. Wang, Z. Gu, and H. Zeng, "Global fixed priority scheduling with preemption threshold: Schedulability analysis and stack size minimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3242–3255, 2016.