Proceedings of the Third International Conference on Machine Learning and Cybernetics, Shanghai, 26-29 August 2004

HYBRID EARLIEST DEADLINE FIRST /PREEMPTION THRESHOLD SCHEDULING FOR REAL-TIME SYSTEMS

DONG-ZHI HE[1], FEI-YUE WANG[1, 2], WEI LI, XIANG-WEN ZHANG

[1] The Key Laboratory of Complex Systems and Intelligence Science, Institute of Automation, Chinese Academy of Sciences, Beijing 100080, China

[2] The Department of Systems and Industrial engineering of the University of Arizona, Tucson, Arizona 85721 USA

E-MAIL:shedz@sina.com

Abstract:

As embedded systems geting increasingly complex, preemption overheads become a serious load problem for many microchip-based application specific systems, and sometimes may even jeopardize the system schedulability. This paper presents a dynamic preemption threshold scheduling (DPT) that integrates preemption threshold scheduling into the earliest deadline first. The DPT scheduling can effectively reduce context switching by threads assignment and changing task dynamic preemption threshold at runtime. Meanwhile, because the algorithm is based on dynamic scheduling, it can achieve higher processor utilization with relatively low costs in preemption switching and memory requirements. The DPT scheduling can also perfectly schedule a mixed task set with preemptive and non-preemptive tasks, and subsumes both as special cases.

Keywords:

Preemption threshold scheduling; earliest deadline first; application specific operating systems; real-time systems; dynamic scheduling; threads

1. Introduction

Real-time systems are a type of systems whose perfect control depends on not only correct calculating results but also the completing time of control flows, that is, each control flow must complete before specified time constraints. Since Liu & Layand proposed rate-monotonic (RM) and earliest deadline first (EDF) scheduling in their classical work [1], the study on preemptive scheduling is almost equivalent to the study on real-time scheduling algorithm. Even it is believed that kernel mechanisms of real-time systems must have preemptive function. Though preemptive schedulers have more advantages than non-preemptive schedulers, such as higher CPU utilization, flexible scheduling, excessive context switching overheads and more memory requirements and are also increased at runtime that undermine these advantages.

In the recent decades, there appears a new trend, application specific operating systems (ASOS), in embedded systems developing [2,3]. ASOS is oriented specific application and suits web development, which often belongs to systems on chip (SOC). One key theme of ASOS is to provide higher performance and lower cost. Hence, it demands that resources of both hard and soft ware of the system are reconfigurable and reusable from design to runtime.

According to the basic demands of SOC, in this paper we present a novel scheduling algorithm, named dynamic preemption threshold (abbreviated DPT) scheduling, which integrates preemption threshold scheduling (PTS) into the EDF. The DPT algorithm can perfectly schedule a mixed task set with preemptive and non-preemptive tasks, and subsumes both as special cases. Thus it remains the scheduling flexibility and higher processor utilization, and also decreases unnecessary context switching and memory requirements at runtime.

The rest of the paper is organized as follows. Section 2 introduces some previous related work. Section 3 presents our scheduling model. Section 4 contains the detail on how to calculate preemption threshold. Section 5 ends the paper with some concluding remarks.

2. Related work

In applying scheduling theory to practice, Burns & Wellings observed the impact of context switching to preemptive scheduling and gave task execution diagram with overheads (see Figure 1). From Figure 1, it is easy to see that context switching overheads become significant when multi-tasking incurs or the task granularity is small. These costs may jeopardize the system schedulability.

To decrease the multi-context-switching, a scheduling with preemption threshold (PTS) was presented in [4,5].

0-7803-8403-2/04/$20.00 ©2004 IEEE

According to the PTS, each task T_i is assigned a fixed

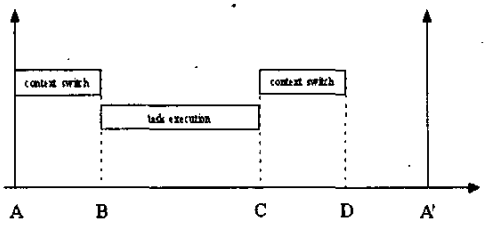

Figure 1. Task execution time with overheads

basic priority π_i by an optimal priority assignment algorithm and a preemption threshold ρ_i with $\rho_i \geq \pi_i$. When task is not executing, its priority is equal to its basic priority; and when it is under execution, its priority simultaneously raises to its preemption threshold. In other words, when a task wants to interrupt another executing task, its basic priority must be higher than the preemption threshold of the executing task. This mechanism has been successful in implementing in the SSX kernel (from REALOGY) and the ThreadX kernel (from Express Logic). In essence, the PTS is a dual priority algorithm, which can automatically generate an implementation model with multi-thread from a design model [4]. It is easy to provides an effective approach to automatic implementation of ROOM-based (Real-time Object Oriented Modeling) designs using PTS [6,7]. Whereas since the PTS is based on static priority scheduling, the processor utilization can not be too higher. For example, there is a task set characterized in Table 1, which utilization is 100%. Under PTS algorithm, Task3 can not meet its deadline no matter how to raise its preemption threshold.

Table 1. The definition of a task set

Name	Execution	Period	Relative Deadline	Stack (bytes)
Task1	1	5	5	20
Task2	3	9	9	40
Task3	3	18	18	40
Task4	6	20	20	70

Using EDF scheduling the task set is schedulable, but the amount of context switching and the memory requirements are increased [see Figure 2.A]. However, if Task1 and Task2 are non-preemptive, the task set is still schedulable and the overheads and the memory requirements are accordingly decreased [see Figure 2.B]. Note that the cost at runtime can be further reduced. Our fundamental motivation is to develop a scheduling algorithm based EDF, which can achieve higher processor utilization in comparison with the static scheduling and meanwhile minimizes the context switching.

Having observed the similarity between the stack resource policy and PTS, Gai et al. presented stack resource policy with threshold (SRPT)[8]. The SRPT gives an approach to transform a static model to dynamic model seamlessly. From above analysis and Figure 2, it is easy to see that the reduction of task preemptions accompanies with the reduction of memory requirements. The goal of SRPT is just for minimizing RAM memory requirements, so the overheads may not be minimal.

The scheduling presented here extends PTS and SRPT at many aspects. First, the DPT scheduling can achieve greater processor utilization than PTS, theoretically even up to all of a processor capacity. Second, the mechanism of the DPT works by the comparison between preemption threshold and preemption level of various tasks; however, PTS do it by the difference between preemption threshold and the basic priority of tasks. Third, in contrast to SRPT, the main goal of the DPT is to minimize context switching instead of achieving the smallest stack space in SRPT. The preemption threshold in the DPT is changeable unlike that in PTS and SRPT, which is fixed at the whole runtime.

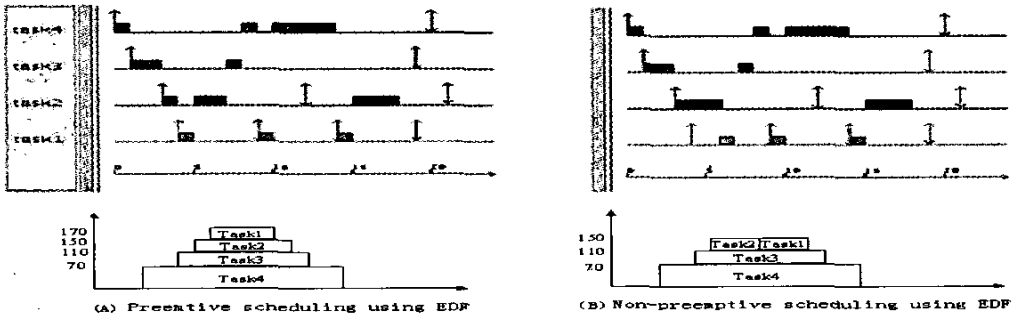

Figure 2. The difference between preemptive and non-preemptive scheduling

3. The model

3.1. Basic terminology

The key theme of a scheduling is to provide a group of rules that determine which task can be executed at each moment to meet its own time constraints. In essence, a task is a series of instructions to complete a relative independent function. A task, T_i, can be characterized by 4-tuple (S_i, C_i, D_i, P_i), where S_i is the release time, C_i is the maximum execution time each of its cycles, D_i is its relative deadline, and P_i is a constant interval between requests for periodic tasks and a minimum interval between request for sporadic tasks. To distinguish from the relative deadline, d_i is used to refer to an absolute deadline. The task set Ω, $\Omega = \{T(C_i, D_i, P_i) : 0 \le i \le n; n \in \mathbb{N}\}$, consists of n independent tasks. Tasks of real-time systems are characterized with stringent timing constraints, that is, each task must meet its deadline. A system is said schedulable if all deadlines of tasks requests are met.

A job is an instance of a task, i.e., a request of the task. We denote the k^{th} request of task T_i by $J_{i,k}$, i.e., the k^{th} job. If t_k and t_{k+1} are the release time of jobs $J_{i,k}$ and $J_{i,k+1}$ respectively, then task T_i is period task when $t_{k+1} = t_k + P_i$; and task T_i is sporadic task when $t_{k+1} \ge t_k + P_i$. A task must be in one of three states at any runtime of a processor: passive, prepared and executing. The passive denotes that the task hasn't been released yet, or it has already completed its current period's workload. The prepared denotes that the task has been released, and has not started execution of its current period's workload. The executing means that the task has captured CPU, that is, it is under execution. Our task model is periodic or sporadic and is scheduled on uni-processor.

3.2. Dynamic preemption threshold policy

In this section, we describe the dynamic preemption threshold policy in terms different from PTS proposed by Saksena & Wang [6, 11]. The dynamic preemption threshold mechanism changes preemption level, rather than the basic priority π_i in various task states to determine which task is executed currently. The mechanism is elaborated as follows:

First, each task T_i is given a basic priority π_i online using EDF; meanwhile, every task is assigned a preemption level φ_i which is inversely proportional to the relative deadline D_i, i.e. $\varphi_i \propto 1/D_i$; in addition, every task is assigned preemption threshold ρ_i with $\rho_i \ge \varphi_i$, of which calculation includes initial and dynamic preemption thresholds. The various jobs of a same task have the identical preemption level and identical initial preemption threshold, but the dynamic preemption thresholds are different. In our model assume that time is discrete and is indexed by the natural numbers.

Each task is assigned different basic priority π_i at different run time by EDF. Because EDF is optimal for synchronous and asynchronous tasks, the dynamic preemption threshold can optimally assign task priority online. Unlike the PTS assigns a fixed priority for each task by another optimal algorithm. If a task is in the passive or prepared states, the decision whether to be scheduled depends its preemption level, whereas if it is executing, its preemption threshold works. For instance, if task T_j wants to preempt task T_i, these conditions, $\pi_j > \pi_i$ and $\varphi_j > \rho_i$, must be satisfied. The FCFS (first come first service) breaks the identical deadline tie of tasks.

Theorem 1: A task set $\Omega = \{T_i(C_i, P_i) : 1 \le i \le n\}$ sorted in non-increasing order by preemption level is schedulable under dynamic preemption threshold scheduling, if it satisfies condition (1) and (2).

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \le 1; \tag{1}$$

$$\forall i, 1 \le i \le n; \forall L, P_k \le L < P_i$$

$$L \ge \sum_{j=1}^{i-1} \left\lfloor \frac{L}{P_j} \right\rfloor C_j + \eta_i \tag{2}$$

Where

$$\eta = \begin{cases} C_i - 1 & ; \exists k, 1 \le k \le i; \rho_i \ge \varphi_k \\ 0 & ; others \end{cases}$$

The proof is skipped over for limit of the paper length.

3.3. Thresholds assignment

The introduction of thread is a useful performance to create ROOM-based implementation models [6,7]. Now we describe a definition which is tightly related to the assignment of thread.

Definition 1: Under dynamic preemption threshold, task T_i and T_j are mutually non-preemptive if $\varphi_i \le \rho_j$ and $\varphi_j \le \rho_i$.

A thread is a subset of a task set within which all tasks must be mutually non-preemptive. To find a method to

partition mutually non-preemptive tasks into a thread is called thread assignment. If the numbers of threads are minimal under a assignment rule, the assignment method is believed to be optimal. Imitating the assignment of minimum number of thread in [4,5], we present a thread assignment for dynamic preemption threshold, named CreateMinThread[Figure.3]. The CreateMinThread algorithm is different from the thread assignment in PTS: The latter assigns threads by the comparison between threshold and basic priority of tasks from low to high threshold, whereas the former works by the difference between preemption level and preemption threshold from high to low preemption level. Thus, it is convenient for calculating preemption power in dynamic threshold calculation. The algorithm assumes that all preemption threshold of the task set are calculated. The task set is sorted in non- increasing order by the initial preemption level. From the first task, mutually non-preemptive tasks are assigned the same thread until no tasks remain the sorted list. Algorithm Assign-Thread is optimal (proof is similar to that in [4,5]).

```
//Create Minimum Number Threads
void CreateMinThread(TaskSet Ω)
{
    ThreadIdex i = 0;
    ThreadGroup G[MaxNum];
    ThreadList L;
    //sort the task set by non-increasing order by preemption Level
    L = SortTaskbyLevel(Ω);
    while(L != 0){
        T_h = FirstTask(L);
        G[i] = {T_h};
        L = RemoveTask(L,T_h);
        for_each T_j ∈ L {
            if(ρ_j ≥ φ_h){
                G[i] = G[i] + T_j;
                L = RemoveTask(L,T_j);
            else i = i + 1;
        }
    }
}
```

Figure. 3 CreateMinThread algorithm for finding minimum number thread

4. Calculating threshold

In this section we will describe preemption threshold calculations that are the key parts for DPT scheduling. Preemption threshold calculation consists of two partitions: initial preemption threshold calculation and dynamic preemption threshold calculation. We will elaborate the calculations in the following part of this section.

4.1. Initial preemption threshold calculation

The initial preemption threshold belongs static priority and is calculated "offline" by the systems. The algorithm for calculating initial preemption threshold works as follows (see Figure 4):

Step 1: To sorted all tasks in non-increasing order by their preemption level and let $\rho_i = \varphi_i$.

Step 2: To test the schedulability of the task set using condition (1).

Step 3: To raise the task preemption threshold starting from the last task T_n and to test the schedulabilty of the task set using condition (2) until the condition is not satisfied. The final preemption threshold is equal to the last preemption threshold that satisfies condition (2).

```
//Calculating Initial Preemption Threshold
void CalculateInitThreshold(TaskSet Ω)
{
    for_each T_i ∈ Ω { ρ_i = φ_i;}
    //sort the task set by non-increasing order by preemption Level
    L = SortTaskbyLevel(Ω);
    if(Utilization(Ω) ≤ 1){
        while (L != 0){
            T_n = LastTask(L);
            m = n;
            while (m != 0){
                if( DemandCPU(P_m) ≤ P_m ){
                    ρ_n = φ_m;
                    m = m-1;
                }
                else{
    //calculating preemption energy for dynamic preemption threshold
                    ε_{n,m} = DemandCPU(P_m) - P_m;
                    m = m-1;
                }
            }
            L = RemoveTask(L,T_n);
        }
    }
    else { return failure; }
}
```

Figure 4. The algorithm of calculating initial preemption threshold

4.2. Dynamic preemption threshold calculation

Except a 4-tuple description for a task, each job of a

task is defined by 2-tuple $(\gamma_{i,k},\theta_{i,k})$, where $\gamma_{i,k}$ is the release time of job $J_{i,k}$, i.e. the k^{th} release time of task T_i; $\theta_{i,k}$ is the earliest time, relative to the release time $\gamma_{i,k}$, that job $J_{i,k}$ is scheduled.

In step 3 of calculating initial preemption threshold, if condition (2) is not satisfied when the preemption threshold of T_i is raised to φ_h, where $h<i$, let $\xi_{i,h}=d_{0,P_h}-P_h$, which is named preemption energy of task T_i to T_h. If two mutually non-preemptive tasks are partitioned into different threads by a thread assignment algorithm, the preemption energy between them is equal to zero. Only tasks belonging different threads need to calculate preemption energy. In the worst case, $\frac{n(n-1)}{2}$ numbers of preemption energies need to be calculated. The preemption energy acts an important part of applying dynamic preemption threshold mechanism. The work of calculating dynamic preemption threshold is elaborated as follow.

Suppose that there are two tasks T_j and T_i come from different threads and $\varphi_j>\rho_i$, that is, T_j may preempt T_i. When the k^{th} job of task T_i, $J_{i,k}$ $(\gamma_{i,k},\theta_{i,k})$, is executing, the h^{th} job of T_j, $J_{j,h}(\gamma_{j,k},\theta_{j,k})$, is released, i.e. $\gamma_{j,h}>\gamma_{i,k}+\theta_{i,k}$. To test the inequality $\gamma_{j,h}-(\gamma_{i,k}+\theta_{i,k}+1)\geq\xi_{i,j}$? If the test of the inequality is true, the preemption threshold of T_i is raised to the preemption level of T_j, i.e. $\rho_i=\varphi_j$. In other words, T_j can not preempt T_i. Inversely, if the result of the test is false, the preemption threshold of T_i is unchanged and until the job $J_{i,k}$ completes. In other words, if a job of task T_i is preempt one time, the preemption threshold is unchanged in the same job.

As is mentioned above, it is easy to see that when the preemption threshold of each task is always equal to its preemption level, the algorithm degenerates to EDF scheduling, and that when the preemption threshold is raised to the highest preemption level, it is equivalent to non-preemptive scheduling.

Though the task set of Table 1 is not schedulable with PTS, it can schedule using DPT since its total utilization is equal to 1. Now to calculate each task's preemption level, preemption threshold and preemption energy in Table 2.

Table 2. Values of preemption level, preemption threshold and preemption energy

Name	φ_i	ρ_i	$\xi_{i,j}$
Task1	4	4	-
Task2	3	4	-
Task3	2	4	-
Task4	1	3	$\xi_{4,2}=0$ $\xi_{4,3}=0$ $\xi_{4,1}=1$

The threads assigned by CreateMinThread algorithm are G1={Task1, Task2, Task3} G2={Task4}. If the preemption between tasks is determined purely by initial preemption thresholds, that is, tasks is mutually non-preemptive in the same thread, the Task4 can be easily interrupted by the other tasks in another thread (see Figure 5.A). However, if to add dynamic preemption threshold factor, Task4 will not be preempted Task2 and Task3, and whether to be preempted by Task1 by the value of preemption energy at run time. From the Figure 6 we know

$$\gamma_{1,3}=10,\ \gamma_{4,1}=8,\ \theta_{4,1}=0$$

It follows

$$\gamma_{1,3}-(\gamma_{4,1}+\theta_{4,1}+1)\geq\xi_{4,1}$$

Hence, under DPT scheduling, the preemption threshold of job $J_{4,1}$ is raised to 4, and Task1 can not preempt it, accordingly reducing a time task switching (see Figure 5.B).

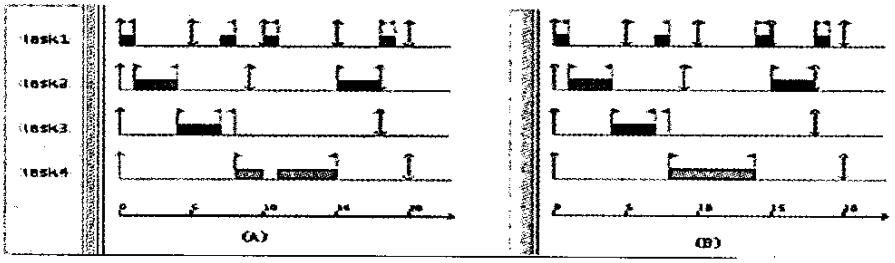

Figure 5. Two different schedules for the same task set: (A) pure thread; (B) dynamic preemption threshold.

5. Conclusions

With the rapidly developing of ASOS based on system on chip, the preemption overheads that are contributed by multi-tasking become non-trivial. The DPT scheduling can reduce the preemption by two way: thread assignment and threshold re-calculation at runtime. The algorithm also ensures that mutually non-preemptive tasks that are partitioned different threads are still mutually non-preemptive at runtime. The DPT scheduling can achieve higher utilization with low runtime cost than PTS. The DPT algorithm also provides a new way to transform static scheduling to dynamic scheduling seamlessly.

Finally, we note that the study on the algorithm is needed to go further in the future work. Because many application specific systems for complex software are applied to uncontrolled environment, the robust DPT scheduling should be provided.

Acknowledgements

This paper is supported in part by the Oversea Outstanding Talent Program from the State Planning Committee and the Chinese Academy of Sciences (under grant No.[1999]0359), the National Outstanding Young Scientist Research Awards(under grant No. 60125310).

References

[1] C.L.Liu and J.W.Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the Association for Computing Machinery, 1973,20(1): 46-61.

[2] F.Y.Wang, Z.H.Wu. ASOS: A developing trend of embedded operating systems. The journal of Chinese Computer World, 2000, (45).

[3] D.Z.He, Z.X.Wang, W.Li. A Scheduling Algorithm for ASOS and its Application to Traffic Control. In Proceedings of IEEE International Conference on Intelligent Transportation Systems, 2003:861-866.

[4] Manas Saksena and Yun Wang.Scalable Real-Time System Design Using Preemption Thresholds. In Proceedings of the IEEE RTSS, 2000.

[5] Y.Wang, M.Saksena. Scheduling fixed-priority tasks with preemption threshold. In Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications (December 1999).

[6] S.Kim,S.Hong and T.H.Kime. Scenario-based implementation architecture for Real-Time Object-Oriented models. In proceedings of International Workshop on Object-Oriented Real-Time Dependable Systems, 2002.

[7] M.Saksena and P.Karvelas. Designing for Schedulability Integrating Schedulability Analysis with Object-Oriented Design. In Proceedings of Euro-Micro Conference on Real-Time Systems, June 2000.

[8] P.Gai, G.Lipari, M.D.Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip. In Proceedings of the IEEE RTSS, 2001:73-83.