A Hybrid Priority Multiprocessor Scheduling Algorithm

Chiahsun Ho and Shelby H. Funk
Computer Science Department
University of Georgia
Athens, GA, USA
{*ho,shelby*}*@cs.uga.edu*

Abstract

We propose a hybrid scheduling algorithm using deferred preemption with an online adjustment. Our approach is to prevent jobs from preempting one another until they have been promoted. We use the tasks' Worst Case Response Time (WCRT) to determine when a promotion must occur in order to ensure safe execution. During execution, this promotion time is adjusted to extend nonpreemptive execution of lower priority tasks whenever possible. Promoted tasks have the higher priority and are scheduled using a fixed-priority scheduling algorithm. Scheduling decisions for non-promoted tasks are more flexible. We consider four strategies for determining which task to preempt: fixed-priority, maximum remaining execution time, least laxity and latest promotion time. Experimental results demonstrate that these strategies successfully avoid most preemptions compared to standard fixed-priority scheduling.

1. Introduction

In real-time systems, correct behavior depends not only upon logical correctness but also upon temporal correctness. In these systems, all jobs have deadlines. In a hard real-time system, each job must be completed no later than its deadline. We consider hard real-time scheduling on multiprocessor systems. Many scheduling algorithms for real-time systems are priority-based, meaning each job has an associated priority and the algorithm will always select the highest priority jobs to execute. However, such a scheme may lead to scheduling overheads due to preemptions and migrations. The aim of our research is to adjust multiprocessor fixed-priority scheduling in order to reduce such overheads.

When a job is preempted, its state must be stored to ensure proper operation upon resumption of execution. In addition, the preempting job may cause the contents of the cache to be overwritten, leading to cache misses that would not have occurred without the preemption. Both of these events lead to the system spending extra time on scheduling, which delays the job's completion time. Furthermore, a preempted job may restart on a different processor, which can extend the job's completion time further because the job cannot start executing until all relevant state information has been transferred to the new processor.

Priority-based scheduling algorithms will only preempt an executing job if a higher-priority job arrives. However, this higher-priority job may be able to wait for the lower-priority job to finish executing. As long as no jobs will miss a deadline as a result, there is no need for the job to execute right away. This strategy, called *delayed preemption* or *cooperative scheduling*, has been studied extensively for uniprocessor systems [1], [2]. Recently, some research has been performed for multiprocessor systems as well [3], [4]. These strategies are primarily aimed towards improving the response time of aperiodic jobs. For uniprocessors, deferred preemption has been considered as as strategy for reducing scheduling overheads by Baruah in [5] and Yao, et al., [6]. In choosing between preemptive and non-preemptive scheduling algorithm to balance feasibility and overheads on uniprocessors. Baruah [5] analyzed deferred preemption scheduling with dynamic priority algorithms, such as EDF. In [6], Yao, et al., determined the largest non-preemptive regions for each task under fixed priority scheduling.

This paper considers the degree to which the delayed preemption strategy can reduce overhead caused by preemptions and migrations. Because there exists a method for finding the WCRT of each task [7], we can determine how long arriving jobs can safely wait before they must preempt the executing lower-priority job. Once the job's safe interval has elapsed it will preempt a lower priority job if the one is executing. We call such an event an *unavoidable preemption*. While some preemptions are unavoidable, we found that many preemptions can be avoided using this technique.

If a task incurs an unavoidable preemption, the scheduler must schedule it according to its fixed priority to ensure it completes on time. Once a task's safe interval has elapsed, we say it has been *promoted*. While promoted tasks must adhere to the fixed priority scheduling strategy, we have more flexibility in the way we schedule non-promoted tasks. We consider a number of different scheduling strategies to determine which job should be preempted. Hence, we present a new scheduling method which safely combines known scheduling strategies with the aim of reducing overhead costs. With this in mind, we wish to consider which algorithms will reduce more overhead when used in conjunc-

tion with a fixed-priority deferred preemption algorithm.

2. Model and Definitions

Our goal is to reduce preemptions when scheduling periodic [8] or sporadic [9], [10] tasks. Below we introduce the terms and notation we will use in the following sections.

Periodic tasks: The periodic [8] and sporadic [9], [10] task models have proven very useful for the modeling and analysis of real-time computer application systems. In this model, real-time processes recur at regular intervals. Each periodic or sporadic task T_i is characterized by two parameters — a worst case execution requirement (e_i) and a period (p_i). Accordingly, we will model a real-time system $\tau \equiv \{T_1, T_2, \ldots, T_n\}$ as being comprised of a collection of n periodic or sporadic tasks. Each periodic task T_i generates an infinite sequence of jobs $T_{i,0}, T_{i,1}, \ldots, T_{i,k}, \ldots$.

We denote the arrival time and deadline of a job $T_{i,k}$ to be $a_{i,k}$ and $d_{i,k}$, respectively. A periodic task $T_i = (p_i, e_i)$ generates each job $T_{i,k}$ at time $a_{i,k} = k \cdot p_i$. Each job $T_{i,k}$ needs to execute for e_i units of time by its deadline of $d_{i,k} = (k+1) \cdot p_i$, for all non-negative integers k. We assume a preemptive schedule which permits migration. Thus, jobs generated by higher-priority tasks can interrupt (preempt) a currently executing lower-priority job and the preempted job may restart on any processor. The costs associated with preemptions and migrations are generally assumed to be included in their WCRT.

Delayed preemption: The tasks of τ are indexed according to their priority. Hence, task T_1's jobs have the highest priority and task T_n's jobs have the lowest priority. According to our scheduling strategy, though, a higher-priority task will not preempt a lower-priority task unless not preempting may cause some deadline to be missed. Whenever the scheduler executes a lower priority job while a higher priority job is waiting, we say a *priority inversion* occurs. Priority inversions must be limited because they can cause deadline misses. However, as we shall see, we can use priority inversions to our advantage as well.

Guan, et al., [7] introduced a method for determining each task's WCRT. We let R_i denote task T_i's WCRT and let Λ_i denote its latest defer time. If each job of T_i has its priority promoted at least R_i time units before its deadline, no deadlines will be missed because of the nature of dual priority [1], [2], [3] and [4]. Hence,

$$\Lambda_i = p_i - R_i. \tag{1}$$

For each job $T_{i,k}$, we let $\lambda_{i,k}$ denote $T_{i,k}$'s *latest promotion time*. Thus,

$$\lambda_{i,k} = a_{i,k} + \Lambda_i. \tag{2}$$

As long as the job $T_{i,k}$ is promoted to T_i's fixed priority by time $\lambda_{i,k}$, the system will be scheduled successfully.

3. Deferred Preemption Algorithm

The scheduler executes tasks in a manner similar to other cooperative schedulers [1], [2], [3], [4]. The scheduler must handle promoted tasks and non-promoted tasks differently. When a processor becomes idle, the scheduler will assign tasks to processors as follows.

1) Tasks with promoted priority.
2) Non-promoted tasks suffering a priority inversion (i.e., delayed preemption tasks).
3) Lower-priority non-promoted tasks.

Our primary goal is to reduce the overheads caused by preemptions and migrations. To this end, we delay preemptions as long as possible while still ensuring the system will run safely (i.e., no deadlines will be missed). As described above, we schedule promoted tasks using a fixed priority scheduling strategy. Non-promoted tasks do not have to adhere to such a strategy. We consider three strategies for selecting which non-promoted task to preempt.

- Maximum remaining execution time
- Least laxity, or
- Latest promotion time.

These strategies were selected because we consider these to be the tasks that are most likely to suffer unavoidable preemptions (i.e., these tasks are the least likely to finish executing before other tasks have their priorities promoted). We also consider task systems with two fixed priorities (i.e., a multiprocessor variant of the dual priority scheduling strategy developed by Davis, et al. [1]).

In order for the delayed preemption strategy to work, we must know the WCRT of the periodic tasks. Lehoczky, et al., [11] developed the time demand analysis (TDA) strategy for uniprocessor systems. This method was extended for multiprocessors [12], [13], [14] and [15] as follows

$$R_i = \frac{1}{m} \sum_{k<i} \left(\left\lceil \frac{R_i}{p_k} \right\rceil e_k + e_k \right) + e_i. \tag{3}$$

Recently, Guan, et al., [7] presented an algorithm that finds a better WCRT upper bound for multiprocessors. Their work examines the maximum interference from higher priority jobs more precisely than Equation 3.

We observe that for all these TDA methods, the value of R_i is the sum of T_i's execution time and the maximum amount of time that T_i may have to wait while higher priority tasks execute. We call this maximum waiting time and T_i's interference time. If a job $T_{i,k}$ executes for ξ_i time units before its promotion time, the amount of time it will execute at the promoted priority level is deceased by ξ time units and the worst case interference time cannot be smaller than it would have been if the job had executed for the full e_i time units after promotion. Therefore, the worst case response time of the remaining $(e_i - \xi_i)$ time units cannot have a response time larger than $(R_i - \xi_i)$. With this in

mind, our algorithm adjusts task priority promotion times whenever a non-promoted task executes. Specifically, if $T_{i,k}$ executes for ξ_i time units and is preempted then

$$\lambda_{i,k} \leftarrow \lambda_{i,k} + \xi_i. \quad (4)$$

In other words, we leave the interference part unchanged in equation 3, but shorten the worst case execution time.

The jitter issue due to delay task priority promotion times can be accommodated by Ha and Liu in [16]. Ideally, each task priority promotion times will release every p time units apart. However, our adjustment strategy might cause a task T_i to have consecutive promotion times that are less than P_i time units apart. While this kind of jittery billowier can often cause problems, it does not have any negative consequences for our algorithm. If a task T_i has consecutive promotion times $p_i - x$ time units apart for some $x > 0$, then T_i's demand during that interval is $e_i - x$. Therefore, T_i does not impose extra interference to lower priority tasks. For a more in depth discussion of this issue, please refer to [16].

In the next section we see that this strategy is very successful at reducing preemptions and migrations – i.e., most preemptions are avoidable.

4. Experimental Results

In this section, we compare standard fixed priority scheduling to the strategies described above for a variety of scenarios. Using Baker's method for randomly generating task sets [17], we generated approximately 10,000 task sets with a variety of different characteristics. The number of processors range from 2 to 32 (i.e., 2, 4, 8, 16, and 32), total utilization from 0.25% to 97.5% of m, and maximum utilization from 0.1 to 0.9. We found WCRT for the tasks using the algorithm developed by Guan, et al., [7]. Because task periods can greatly influence the frequency of preemptions, we use three different period sets to generate task sets.

- Harmonic task sets – $P_{i+1} = k \cdot P_i, \forall i = 1$ to $n - 1$
  - Periods of harmonic task sets have been chosen as: {4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048}
- Relatively prime task sets – $gcd(P_i, P_j) = 1, \forall i \neq j$
  - Periods of relatively prime numbers between 3 and 131
- Random task sets – $10 \leq P_i \leq 100, \forall i = 1$ to n
  - Periods of 30 random numbers within [10,100]

We use the following expression to reflect overhead: $w_1 \cdot$ number of preemptions + $w_2 \cdot$ number of migrations, where w_1 and w_2 are weight factors. Because migrations tend to be more time consuming than preemptions, we set $w_1 = 1$ and $w_2 = 3$ for the reported experiments. An average savings of 100% means all of the preemptions and migrations in the fixed-priority schedule were avoidable (i.e., the hybrid schedule executed all tasks non-preemptively); average savings of 0% means either all preemptions and migrations are unavoidable or the fixed-priority schedule had no preemptions or migrations.

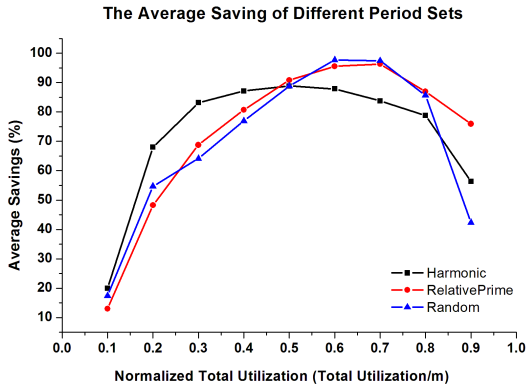

Figure 1: The average savings of different periods (Total Normalized Utilization)

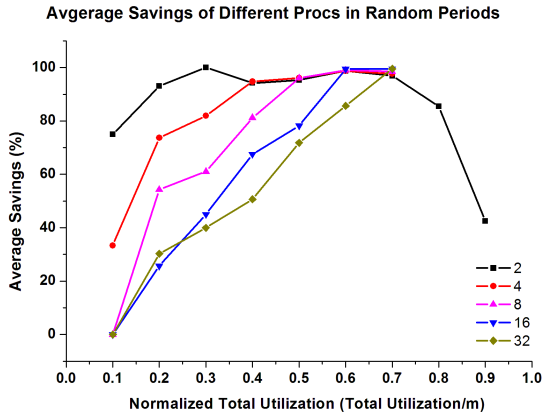

Figure 2: The average savings of different processors in random periods

Figure 1 compares the overhead savings for the different period scenarios. All the strategies perform quite well. When maximum utilization is small, there is no savings because the fixed-priority schedule preempts infrequently. As maximum utilization increases, the savings improve significantly. When maximum utilization is 30%, over half the preemptions and migrations are avoided. When it reaches 50%, over 80% of the overhead is avoided.

Figure 2 illustrates the average savings for each of our processor scenarios considering the random period scenario. The other scenarios have similar results. For more than two processors, we were unable to generate feasible task sets with utilization over 70%. Notice that we got higher savings when there were fewer processors. We believe that this is because having more processors increases the likelihood that some jobs can finish before a priority promotion occurs.

5. Conclusion

We consider a hybrid scheduling strategy using delayed preemption of fixed-priority scheduling in conjunction with four strategies for selecting which task to be preempted. We use WCRT to determine when a preemption is unavoidable. During execution, we adjust this value if a task is able to execute prior to its promotion time. Experimental results indicate that these strategies can reduce a significant number of preemptions and migrations.

In future we plan to consider other hybrid algorithms to see if they do equally well. We are particularly interested in reducing the preemptions and migrations in optimal scheduling algorithms such as Pfair [18], LLREF [19], BF [20] and DP-Wrap [21]. We also plan to study using the deferred preemption strategy to schedule infeasible task sets.

References

[1] R. Davis and A. Wellings, "Dual priority scheduling," in *IEEE Real-Time Systems Symposium (RTSS)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 100.

[2] R. Jejurikar and R. Gupta, "Procrastination scheduling in fixed priority real-time systems," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 2004, pp. 57–66.

[3] A. Tumeo, M. Branca, L. Camerini, M. Ceriani, G. Palermo, F. Ferrandi, D. Sciuto, and M. Monchiero, "A dual-priority real-time multiprocessor system on FPGA for automotive applications," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 1039–1044.

[4] A. A. Josep M. Banus and J. Labarta, "Extended global dual priority algorithm for multiprocessor scheduling in hard real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005.

[5] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," *IEEE Real-Time Systems Symposium (RTSS)*, vol. 0, pp. 137–144, 2005.

[6] G. Yao, G. Buttazzo, and M. Bertogna, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009, pp. 351 –360.

[7] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," *IEEE Real-Time Systems Symposium (RTSS)*, vol. 0, pp. 387–397, 2009.

[8] 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.

[9] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[10] M. Dertouzos, "Control robotics : the procedural control of physical processors," in *Proceedings of the IFIP Congress*, 1974, pp. 807–813.

[11] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *IEEE Real-Time Systems Symposium (RTSS)*. Santa Monica, California, USA: IEEE Computer Society Press, Dec. 1989, pp. 166–171.

[12] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition," *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, vol. 0, p. 337, 2000.

[13] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 3rd ed. Addison-Wesley, 2001.

[14] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 149 –160.

[15] L. Lundberg, "Multiprocessor scheduling of age constraint processes," in *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Washington, DC, USA: IEEE Computer Society, 1998, p. 42.

[16] R. Ha and J. W.-S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*. Los Alamitos: IEEE Computer Society Press, Jun. 1994.

[17] T. P. Baker, "An analysis of fixed-priority schedulability on a multiprocessor," *IEEE Real-Time Systems Symposium (RTSS)*, vol. 32, no. 1-2, pp. 49–71, 2006.

[18] S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, June 1996.

[19] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, Los Alamitos, CA, USA, 2006.

[20] D. Zhu, D. Mosse, and R. Melham, "Multiple-resource periodic scheduling problem: how much fairness is necessary?" in *IEEE Real-Time Systems Symposium (RTSS)*, December 2003, pp. 142–151.

[21] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010, pp. 1–10.