An Optimal Discrete-Time Based Boundary Fair Scheduler for Sporadic Tasks in Multiprocessor Real-Time Systems*

Yifeng Guo, Hang Su and Dakai Zhu
The University of Texas at San Antonio
San Antonio, TX 78249
{yguo, hsu, dzhu}@cs.utsa.edu

Abstract

Several optimal scheduling algorithms have been studied recently for various real-time tasks running on multiprocessor systems with continuous *as well as* discrete *time. However, the existing optimal schedulers for sporadic tasks may incur very high scheduling overhead due to either excessive number of scheduling points (e.g., for the Pfair-like scheduler) or arbitrarily small allocation for tasks (e.g., for the continuous-time based DP-Fair scheduler). In this paper, extending the* boundary fair (Bfair) *scheduler that is optimal for periodic tasks, we investigate an optimal scheduling algorithm for such* sporadic tasks with implicit deadlines *in multiprocessor systems. First, considering the irregular arrivals of sporadic tasks, we discuss how to determine the next boundary time (i.e., the deadline of a task) to minimize the scheduling points. Then, an example is presented to illustrate the major scheduling steps that incorporate the dynamic arrivals of sporadic tasks, followed by the formal presentation of the scheduling algorithm. We point out our future work in the end.*

1 Introduction

Numerous scheduling algorithms have been proposed in the last few decades to guarantee various hard and/or soft timing constraints for different (e.g., periodic, sporadic and aperiodic) real-time tasks, on both single- and multiprocessor systems. Although the scheduling theory for uniprocessor systems has been well developed, the scheduling for multiprocessor real-time systems is still an evolving research field and many problems remain open due to their intrinsic difficulties. With the emergence of multicore processors, there is a reviving interest in scheduling algorithms for multicore/multiprocessor real-time systems and many interesting results have been reported recently.

In this work, we focus on optimal scheduling algorithms for multiprocessor real-time systems, which can achieve full system utilization while guaranteeing the schedulability of tasks. As the first well-known of such scheduling algorithms, the *proportional fair (Pfair)* scheduler was studied for a set of periodic real-time tasks with quantum-based timing requirements [3]. The central idea of Pfair is to enforce proportional progress (i.e., *fairness*) for all tasks by ensuring that the scheduling error from the ideal fluid schedule is within one unit at *each and every* time unit. Several variations of Pfair have been studied to improve its performance [2, 4] or to handle sporadic tasks [1]. Note that, by making scheduling decisions at each quantum time point, Pfair schedulers can incur very high scheduling overhead.

For systems with *continuous* time, the *T-L Plane* based scheduling algorithms have been studied, which make scheduling decisions at the arrival time or deadline of tasks [5]. Recently, a generalized deadline-partitioned fair (*DP-Fair*) scheduling model was investigated, again for continuous time based systems [6]. Although DP-Fair can handle *sporadic* real-time tasks, the continuous time model requires to allocate time share for every active task within *any* interval (i.e., between adjacent deadlines of tasks), which can lead to arbitrarily small allocation and thus result in high scheduling overhead (e.g., context switches).

Observing the fact that a periodic real-time task can only miss its deadline at its period boundary, we have previously studied an optimal discrete-time based *boundary fair (Bfair)* scheduling algorithm, which makes scheduling decisions and ensures fairness for tasks *only* at their period boundaries (i.e., deadlines of tasks) [7]. That is, at each period boundary, the allocation error for any task is less than one time unit. We have shown that, compared to Pfair schedulers, Bfair can significantly reduce the number of scheduling points and associated scheduling overhead. However, the old Bfair scheduler cannot handle sporadic tasks due to their irregular arrival patterns.

*This work was supported in part by NSF awards CNS-0720651, CNS-0855247, CNS-1016974 and NSF CAREER Award CNS-0953005.

2 System Models

We consider a set of n sporadic real-time tasks, $\Gamma = \{T_1, \ldots, T_n\}$, where each task $T_i = (c_i, p_i)$ is characterized by its worst case computation requirement c_i and *minimum* inter-arrival time p_i. Here, p_i is also T_i's relative deadline. That is, we consider sporadic tasks with *implicit deadlines*. Moreover, both c_i and p_i are assumed to be integer multiples of a system unit time and $c_i \leq p_i$. The weight/utilization of task T_i is defined as $w_i = \frac{c_i}{p_i}$, and the system utilization is $U = \sum_{i=1}^{n} w_i$. We further assume that $U = m$, where m is the number of available processors.

When an instance of task T_i arrives at time t, it should be allocated exactly c_i time units of a processor within its active interval from t to $t+p_i$, which is represented as $[t, t+p_i)$. Moreover, the allocation of processors to tasks should satisfy the following two constraints [4]:

- *C1:* A processor is allocated to only one task at any time (i.e., processors cannot be shared *concurrently*);
- *C2:* A task is allocated at most one processor at any time (i.e., tasks cannot be executed in *parallel*);

3 Scheduling Points for Sporadic Tasks

Note that, the scheduling points of the Bfair scheduler are tasks' periodic boundaries, which are actually deadlines of task instances. For periodic tasks, the boundary times are fixed as tasks arrive regularly. However, for sporadic tasks, task instances can arrive at *any* time provided that they are separated by the minimum inter-arrival time. Therefore, finding the *next earliest boundary time* (i.e., the earliest deadline) is not straightforward. To minimize the number of scheduling points and reduce the scheduling overhead, we would like to have the next boundary time be as late as possible to get a longer scheduling interval.

At a given boundary time point b_k ($k \geq 0$), we first add the task instances that arrive at time b_k into the ready queue. Then, we can categorize tasks into three different types: a). the *active tasks* are the ones that have a task instance in the ready queue; b). the *early-completion tasks* are the ones whose deadlines for the current task instances are later than b_k but their task instances have finished their executions (i.e., they are *running ahead*); and c). the *delayed tasks* are the ones whose minimum inter-arrival time reached before or at b_k but their task instances have not arrived yet.

For active tasks, their current deadlines are known. For early-completion tasks, their current task instances have finished their executions and will not miss their deadlines. Therefore, we only need to consider the next task instance for such tasks that can arrive no early than the deadline of their corresponding current task instances. For delayed tasks, their task instances can arrive as early as $(b_k + 1)$. Note that, for both early-completion and delayed tasks, assuming that their next task instances arrive at the earliest possible time, we can get their *expected* deadlines.

Therefore, taking all tasks into consideration, the next boundary time b_{k+1} should be the earliest deadline of task instances in the ready queue as well as those that are expected to arrive in the future. More formally, we have:

$$b_{k+1} = \min\{d_i | T_i \in \Gamma\} \quad (1)$$

$$d_i = \begin{cases} d_i^c & T_i \in \Phi_k \\ d_i^c + p_i & T_i \in \Psi_k \\ (b_k + 1) + p_i & T_i \in \Omega_k \end{cases} \quad (2)$$

$$\Gamma = \Phi_k \cup \Psi_k \cup \Omega_k \quad (3)$$

where d_i^c is the deadline of T_i's current task instance. Moreover, Φ_k, Ψ_k and Ω_k represent the sets for active, early-completion and delayed tasks, respectively, at time b_k. After obtaining the next boundary time b_{k+1}, the enhanced Bfair scheduler will allocate processors to all tasks and generate the initial *schedule slice* for the time interval of $[b_k, b_{k+1})$.

4 An Example

First, we illustrate the major scheduling steps through a concrete example. Here, the task set consists of seven sporadic tasks: $T_1 = (2, 5)$, $T_2 = (3, 15)$, $T_3 = (2, 6)$, $T_4 = (20, 30)$, $T_5 = (6, 30)$, $T_6 = (8, 10)$, and $T_7 = (4, 10)$. Note that, $\sum_{i=1}^{7} w_i = 3$ and 3 processors are assumed.

Suppose that the task instances arrive as follows: two instances of T_1 arrive at time 0 and 6; one instance of T_2 at time 1; two instances of T_3 at time 2 and 10; and one instance of T_4, T_5, T_6 and T_7 at time 2, 4, 4 and 6 respectively. At the initial boundary time $b_0 = 0$, after adding the first task instance of T_1 into the ready queue, we have $\Phi_0 = \{T_1\}$, $\Psi_0 = \emptyset$ and $\Omega_0 = \{T_2, T_3, T_4, T_5, T_6, T_7\}$. From Equation (1), we can find that the next boundary time is $b_1 = 5$, which is the deadline of T_1's first task instance.

As in the Bfair scheduler, to guarantee all deadlines are met, *all* tasks should make appropriate progress. Therefore, when allocating processors to tasks for the time interval $[b_0, b_1) = [0, 5)$, every task needs to get at least its *mandatory* units, which can be calculated as follows: For active tasks, their mandatory units can be obtained in the same way as in Bfair; For early-completion and delayed tasks, assuming their next instances arrive at their earliest expected time, their worst case (maximum) mandatory units can be calculated accordingly as well. Moreover, after allocating mandatory units, a certain number of optional units should be allocated to eligible tasks and ensure the accumulate remaining work of all tasks is no more than 0.

For the above example, after allocating both mandatory and optional units for the first interval $[0, 5)$, the tasks get 2, 1, 1, 3, 1, 4 and 1 unit(s), respectively. Note that, for

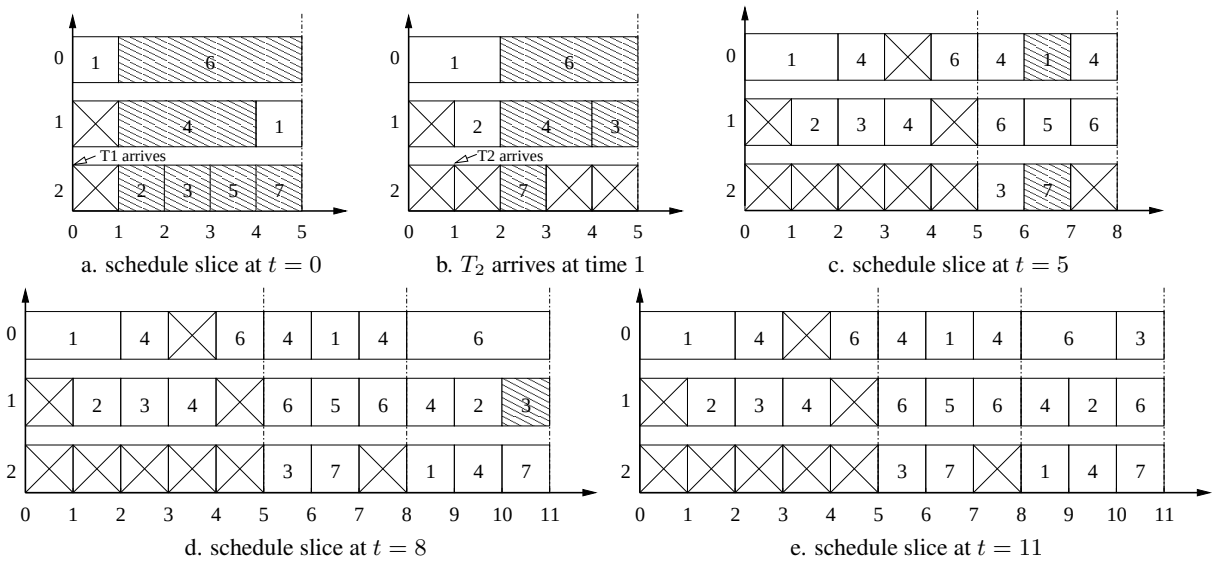

Figure 1. Example: sporadic task scheduling within two scheduling slices

active task T_1, its allocation is *real* and will be consumed. However, for other delayed tasks, their allocated units are only *reserved*, which may be adjusted if they do not arrive as expected. After packing tasks' allocation to processors following a certain mechanism, the initial schedule slice is shown in Figure 1(a). Here, the *real* allocation units are represented by blank rectangles and *reserved* units are represented by shaded rectangles. Moreover, when processors need to be idle due to, for instance, the lack of active tasks, the corresponding rectangle is marked with a 'X'. After the instance of T_2 arrives as expected at time 1, the adjusted schedule is shown in Figure 1(b), where T_2's allocation becomes to be real while other tasks' allocations are adjusted accordingly assuming that they would arrive at time 3.

At time $b_1 = 5$, after the instances of T_3, T_4 and T_6 arrive and being partially executed, we can find that the next boundary time is $b_2 = 8$ and the schedule slice is shown in Figure 1(c). Figures 1(d) and 1(e) shows the schedules at time 8 and 11, respectively. Note that, for simplicity, we limit the actual executed units of tasks to be their proportional allocation within each interval. Therefore, although there are active tasks with unfinished execution times, processors can still be idle as shown in the schedules.

5 Sporadic-BFair Algorithm

Algorithm 1 summarizes the major steps of the *Sporadic-Bfair* scheduler. In the algorithm, same terms have been used as in the old Bfair algorithm [7]. At the very beginning, the task sets are updated with newly arrived tasks being added to Φ_k and the next boundary time b_{k+1} is obtained (lines 1 and 2). For all tasks, their (*expected*) mandatory units are first allocated (lines 4 to 11). Then, the number of *optional* units that need to be allocated to ensure appropriate progress of all tasks is determined and allocated (lines 13 and 14). Finally, the remaining work of tasks is updated and the initial schedule slice is generated (lines 15 to 19).

Algorithm 1 Sporadic-Bfair Algorithm at time b_k

1: Update the task sets Φ_k, Ψ_k and Ω_k;
2: Get the next boundary time b_{k+1} from Equation 1;
3: /*First, calculate and allocate mandatory units;*/
4: **for** ($\forall T_i \in \Phi_k \cup \Psi_k$) **do**
5: $\quad m_i^{k+1} = max\{0, \lfloor RW_i^k + (b_{k+1} - b_k) \cdot w_i \rfloor\}$;
6: $\quad RW_i^{k+1} = RW_i^k + (b_{k+1} - b_k) \cdot w_i - m_i^{k+1}$;
7: **end for**
8: **for** ($\forall T_i \in \Omega_k$) **do**
9: $\quad m_i^{k+1} = \lfloor (b_{k+1} - (b_k + 1)) \cdot w_i \rfloor$;
10: $\quad RW_i^{k+1} = (b_{k+1} - (b_k + 1)) \cdot w_i - m_i^{k+1}$;
11: **end for**
12: /*Then, find and allocate optional units if needed;*/
13: $OU = \lceil \sum_{i=1}^{n} RW_i^{k+1} \rceil$;
14: $AllocateOptionalUnits(OU)$;
15: /*Finally, update workload and generate schedule slice;*/
16: **for** $(T_1, \ldots, T_n)$ **do**
17: $\quad RW_i^{k+1} = RW_i^{k+1} - o_i^{k+1}$;
18: **end for**
19: $GenerateScheduleSlice(b_k, b_{k+1})$;

5.1 Number of Optional Units and Allocation

From Algorithm 1, we can see that the calculation of mandatory units for tasks is quite straight-forward. How-

ever, due to the late arrival of sporadic tasks, it is inevitable to have some processors be idle (e.g., when there are no enough active tasks). Therefore, different from the Bfair algorithm that allocates all available time units within a given interval, it is not trivial to determine the number of optional units that need to be allocated. With the optional units, we should ensure appropriate progress of all tasks while guaranteeing all allocated (mandatory and optional) units for tasks can be successfully packed to processors.

Note that, after allocating mandatory units, there is $RW_i^{k+1} < 1$ (e.g., it is fair for individual tasks) at time b_{k+1}. However, to ensure that no required workload is postponed to the next time interval, we need to have $\sum_{i=1}^{n} RW_i^{k+1} \leq 0$. That is, the whole task set needs to make proportional progress. Therefore, the number of optional units can be found as $OU = \lceil \sum_{i=1}^{n} RW_i^{k+1} \rceil$, where RW_i^{k+1} is the remaining work of task T_i after it receives its mandatory units.

To ensure that no task is required to run in parallel (the constraint C2), not all tasks are *eligible* for such optional units. A task T_i is said to be *eligible* if it is not *running ahead* (i.e., $RW_i^{k+1} > 0$) and not *fully allocated* (i.e., $m_i^{k+1} < b_{k+1} - b_k$). For eligible tasks to compete the optional units, we have extended the priority with *characteristic value* and *urgency factor* for sporadic tasks and higher priority eligible tasks will get one optional unit each.

5.2 Schedule Slice Generation

After each task gets its mandatory and optional units, generating the schedule slice (i.e., packing tasks' allocation to processors) brings in new issues. First, for delayed and early-completion tasks, we should not pack them to processors before their earliest arrival times. Second, no task should be packed to more than one processors at any given time unit (the constraint C2 in Section 2).

Considering the above requirements, we propose a new sub-interval based packing algorithm to generate the schedule slice, which adopts the *longest task first (LTF)* heuristic and *McNaughton* packing rule [6]. Suppose that there are g different possible arrival times for the delayed and early-completion tasks within the interval $[b_k, b_{k+1})$, which are denoted as $a_1, ..., a_g$ ($g \geq 0$). Moreover, we assume that $a_0 = b_k$ and $a_{g+1} = b_{k+1}$. That is, there are $(g+1)$ sub-intervals. The x's sub-interval is denoted as $[a_{x-1}, a_x)$ ($x = 1, ..., g+1$). As mentioned early, for active tasks, their allocations can be packed to any sub-interval. However, for delayed or early-completion tasks that suppose to arrive at time a_x, their allocations can only be packed to sub-intervals after time a_x.

The packing algorithm will allocate tasks to processors one sub-interval at a time from the first to the last sub-interval. For the x's sub-interval $[a_{x-1}, a_x)$, only the tasks that arrive on or before time a_{x-1} and still have remaining allocated units can be packed to processors. To ensure that there is no parallel execution of tasks during the remaining sub-intervals, we first find the *mandatory packing units* for each task T_i, which will be $\max\{0, r_i - (b_{k+1} - a_x)\}$, where r_i is the number of remaining allocated units for T_i. Then, depending on tasks' remaining allocated units, tasks will be packed to processors in the *LTF* order following the McNaughton rule. Note that, the number of units that are packed to processors for any task should be no more than $(a_x - a_{x-1})$, the length of the x's sub-interval.

5.3 Runtime Schedule Adjustment

Note that, unless the delayed tasks arrive as expected, the initial schedule slice needs to be adjusted when such tasks arrive late. That is, in addition to the *primary scheduling points* at boundary times, a light-weighted on-line scheduler (which is similar to Algorithm 1) needs to be invoked to recalculated the time allocation for newly arrived and other delayed tasks and to adjust the schedule based on the current runtime information about all tasks.

6 Conclusions and Future Works

In this work, we investigate an enhanced Bfair scheduling algorithm for sporadic real-time tasks in discrete-time based multiprocessor systems. By considering the dynamic arrival of sporadic real-time tasks, we discuss how to determine the boundary time (i.e., scheduling points) and present the major steps for the *Sporadic-Bfair* algorithm. For our future work, we will finalize the detailed steps of the algorithm as well as the formal proof on the schedulability and optimality of the proposed scheduling algorithm. Moreover, we will conduct extensive simulations and evaluate its performance in terms of reducing scheduling overhead when comparing to exiting schedulers.

References

[1] James H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *RTCSA*, 2000.

[2] J.H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Proc. of ECRTS*, 2001.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[4] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of IPPS*, 1995.

[5] H. Cho, B. Ravindran, and E.G. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proc. of RTSS*, 2006.

[6] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Proc. of ECRTS*, 2010.

[7] D. Zhu, D. Mossé, and R. Melhem. Periodic multiple resource scheduling problem: how much fairness is necessary. In *RTSS*, 2003.