Online Semi-Partitioned Multiprocessor Scheduling of Soft Real-Time Periodic Tasks for QoS Optimization

Behnaz Sanati, Albert M. K. Cheng
Department of Computer Science, University of Houston, Texas, USA
Emails: {bsanati; acheng}@cs.uh.edu

***Abstract*—*In this paper, we propose a novel semi-partitioning approach with an online choice of two approximation algorithms, Greedy and Load-Balancing, to schedule periodic soft real-time tasks in homogeneous multiprocessor systems. Our objective is to enhance the QoS by minimizing the deadline misses and maximizing the total reward or benefit obtained by completed tasks in minimum response time. Many real-time applications and embedded systems can benefit from this solution including but not limited to video streaming servers, multi-player video games, cloud applications, medical monitoring systems, and IoT.*

Keywords: *Periodic tasks, Quality of service, Partitioning, Multiprocessor scheduling, Approximation algorithms.*

1. INTRODUCTION

Multiprocessor systems are widely used in a fast-growing number of real-time applications and embedded systems. Two examples of such systems are Cloud applications [1] and IoT [2]. In hard real-time systems, meeting all deadlines is critical, while in soft real-time systems, missing few deadlines does not drastically affect the system performance. However, it would compromise the quality of the service.

In such systems, jobs meeting their deadlines will gain a reward (also called benefit). Hence, researchers focus on maximizing rewards to improve the QoS. Besides the total reward, other factors also influence QoS, such as overall response time (makespan plus scheduling time) and deadline-miss ratio. Multiprocessor real-time scheduling algorithms may follow a partitioned or global approach or some hybrid of the two, called semi-partitioning.

Global scheduling can have higher overhead in at least two respects: the contention delay and the synchronization overhead for a single dispatching queue is higher than for per-processor queues; the cost of resuming a task may be higher on a different processor than on the processor where it last executed, due to inter-processor interrupt handling and cache reloading. The latter cost can be quite variable, since it depends on the actual portion of a task's memory that remains in cache when the task resumes execution, and how much of that remnant will be referenced again before it is overwritten [1]. These issues are discussed at some length by Srinivasan *et al.* [3]. Elnably *et al.* [1] study fair resource allocation and propose a reward-based model for QoS in Cloud applications. In contrast, Alhussian, Zakaria and Hussin [4] prefer global scheduling and try to improve real-time multiprocessor scheduling algorithms by relaxing the fairness and reducing preemptions and migrations.

Amirijoo, Hansson and Son [5] discussed specification and management of QoS in real-time databases supporting imprecise computations. Reward-based scheduling of periodic tasks has also been studied by Aydin *et al.* [6], and Hou and Kumar [7]. Aweruck *et al.* [8] proposed a reward-maximizing model for scheduling aperiodic tasks on uniprocessor systems which can also be applied to multiprocessors. We have also previously studied reward-based scheduling of aperiodic real-time tasks on multi-processor systems. We proposed two algorithms, GBBA [9] and LBBA [10], and provided performance analysis and comparative experimental results of those algorithms versus another state-of-the art algorithm [8].

Significant improvements obtained by LBBA method, especially in reducing the overall response time (i.e., scheduling time plus makespan of the task sets), in addition to maximizing the total reward and minimizing tardiness, showed promising enhancement in QoS. That encouraged us to expand our research to solving the problem of scheduling periodic (and sporadic) soft real-time tasks on multi-processor systems, on which relatively very little research has been done. LBBA is using partitioning strategy for aperiodic tasks. Now to extend it for scheduling periodic (and sporadic) tasks, we use semi-partitioning at job boundaries.

Semi-partitioned real-time scheduling algorithms extend partitioned ones by allowing a subset of tasks to migrate. Given the goal of "less overhead," it is desirable for such strategy to be boundary-limited, and allow a migrating task to migrate only between successive invocations (job boundaries). Non-boundary-limited schedulers allow jobs to migrate, which can be expensive in practice, if jobs maintain much cached state.

Previously proposed semi-partitioned algorithms for soft real-time (SRT) tasks such as EDF-fm and EDF-os [11], have two phases: an offline assignment phase, where tasks are assigned to processors and fixed tasks

(which do not migrate) are distinguished from migrating ones; and an online execution phase. In their execution phase, rules that extend EDF scheduling are used. The goal in these strategies is to minimize tardiness.

In this paper, we propose a new online reward-based semi-partitioning approach to schedule periodic soft real-time tasks in homogeneous multiprocessor systems. We use an online choice of two approximation algorithms (Greedy approximation and Load-Balancing) for partitioning, which provides an optimized usage of processing time. In this method, no prior information is needed. Hence, there is no offline phase.

Our objective is to enhance the QoS by minimizing tardiness and maximizing the total reward obtained by completed tasks in minimum makespan. Therefore, we allow different jobs of any task get assigned to different processors (migration at job boundaries) based on their reward-based priorities and workload of the processors. This method can also direct SRT systems with mixed set of tasks (aperiodic, sporadic and periodic) by defining their deadlines accordingly.

Many real-time applications can benefit from this solution including but not limited to video streaming servers, multi-player video games, mobile online banking and medical monitoring systems. For example, consider mobile banking applications that are set to send monthly statements, weekly or daily balance notifications (periodic) and also notifications when a check is posted or the balance is less than specific amount (aperiodic).

Another example is a medical monitoring application installed on a physician´s laptop or smart phone which periodically receives the patients´ vital signs, such as blood pressure, number of heart beat, breathing per minute, etc. from the body sensor networks attached to the patients. It process and records them periodically and in case they go out of range and the situation is critical, sends alert (aperiodic). In the next sections, we explain our novel semi-partitioning hybrid model, which combines reward and cost models, for optimizing quality of service in soft real-time systems.

2. OUR CONTRIBUTION

2.1. System and Task Model

A multiprocessor system with *m* identical processors is considered for partitioned, preemptive scheduling of periodic soft real-time task sets with implicit deadline. Each processor has its own pool (for ready tasks), stack (for preempted and running tasks) and garbage collection (for completed and tasks which missed deadlines). Each periodic task may be released at any time. Tasks are independent in execution and there are no precedence constraints among them. Pre-emption is allowed. A desired property of the system in this method is the possibility to delay jobs without drastically reducing the overall system performance.

2.2. Our Methodology

Semi-Partitioning Model:

This algorithm applies online semi-partitioning. In our partitioning approach, no job migration is allowed. In other words, each job, i.e. an instance of a task, will be assigned to a processor at release time, based on its priority and worst-case execution time, and also the current workloads of the processors, and it has to stay with that processor during its entire runtime in the system. However, different instances of a periodic task may be assigned to different processors. This method is possible since each processor has its own pool for the ready tasks assigned to it.

Online Choice of Approximation Algorithms:

We consider Greedy and Load-balancing approximation algorithms, one of which will be chosen online based on the conditions of the system at each time instance, for partitioning and scheduling task instances in order to optimize the CPU usage, minimize the makespan and prevent starvation of low priority tasks. We explain it in more details in subsection 2.4.

2.3. Definitions

Periodic Tasks:

A *periodic* task, in real-time systems, is a task that is periodically released at a constant rate. Usually, two parameters are used to describe a periodic task T_i; its execution w_i as well as its period p_i. An instance of a periodic task (*i.e* release) is known as a job and is denoted as $T_{i,j}$, where j=1, 2, 3, … . The deadline of a job is the arrival time of its successor. For example, the deadline of the j^{th} job of T_i, which is $T_{i,j}$, would be the arrival time of job $T_{i,(j+1)}$, that is at jp_i.

Notations:

We define the notations used throughout this paper as follows:

$r_{i,j}$ – *release time* of job $T_{i,j}$

w_i – *execution time* of job $T_{i,j}$, simply considered as *workload* of job $T_{i,j}$ in this paper

p_i – *period of task* T_i

$s_{i,j}$ – *start time* of job $T_{i,j}$

$c_{i,j}$ – *completion time* of job $T_{i,j}$

$Br_{i,j}$ – *break point* or *deadline* of job $T_{i,j}$, is the minimum of:

$$Br_{i,j} = \min (p_i \parallel s_{i,j} + 2w_i) \quad (1)$$

$\beta_i(t)$ – *benefit density function* of task T_i at time t, for ($t \geq w_i$), which is a non-increasing, non-negative function, with the following restriction to be satisfied for each $\beta_i(t)$:

$$\frac{\beta_i(t)}{\beta_i(t+w_i)} \leq C \quad (2)$$

Note: for $t < w_i$, there would be no benefit gained by job $T_{i,j}$, since it has certainly not completed its execution at time t.

$f_{i,j}$ – *flow time* of job $T_{i,j}$:

$$f_{i,j} = c_{i,j} - r_{i,j} \quad (3)$$

$b_{i,j}$ – *benefit*, gained by a completed job $T_{i,j}$:

$$b_{i,j} = w_i . \beta_i (f_{i,j}) \quad (4)$$

LBBA Algorithm for Periodic Tasks

```
1  Required: One or more jobs arrive at time t ≥ 0
2  {
```

Job Arrival

```
3    /* TempList: list of ready jobs waiting for
4       distribution among processors */
5
6    Append the arrived job(s) to the TempList
```

Benefit-Based Scheduling

```
7    Calculate the priority of each job T_i,j in the
8      TempList:
9      d_i,j (t) = β_i(t + w_i - r_i,j)
10   Sort TempList based on the priority
11   If (at least one stack is empty)
12   {
13         Push the highest priority job(s) T_i,j
14           onto empty stack(s) of idle processor(s) l;
15         Add its execution time w_i to total workload
16           of the stack of the processor l (∑ Ws_l ),
17         Recalculate total workload of processor l:
18           W_l = ∑ Wp_l + ∑ Ws_l
19         Calculate the fixed priority of j using its
20           start time s_i,j:
21           d'_i,j(t) = β_i(s_i,j + w_i – r_i,j)
22         Start executing j,
23   }
24   Else
25   {
26        /* no stack is empty */
27        /* preempt if possible otherwise distribute
28           among the pools */
29        Compare the priority of the ready jobs in
30           TempList with the priority of the running
31           jobs (indicated by index k) on top of the
32         stacks:
33         If (d_i,j(t) ≤ 4d'_k for ( each job T_i,j in TempList
34           and each running job T_k) )
35         {
36              /* no preemption allowed */
37              /* partition the ready jobs among
38                 pools of the processors */
```

Load-Balancing Approximation (for Partitioning)

```
39              For (each job T_i,j in TempList)
40              {
41                Sort the processors in ascending order of
42                   their total remaining workload on their
43                   pools and stacks:
44                   W_l = ∑ Wp_l + ∑ Ws_l
45                Append the jobT_i,j with largest
46                   execution time w_i to the pool of the
47                   processor l with minimum remaining
48                   work load;   /* load balancing */
49                Remove T_i,j from TempList;
50                Add its execution time w_i to total
51                  workload of the pool of processor l
52                  (∑ Wp_l );
53                Recalculate total workload of
54                  processor l:
55                  W_l = ∑ Wp_l + ∑ Ws_l
56             }
57         }
58        Else
59             /* if (d_i,j(t) > 4d'_k)  then (T_i,j preempts T_k)*/
```

Greedy Approximation (multiple-choice Preemption)

```
60             /* If T_i,j has more than one choice of
61                processors, it will be pushed onto
62                the stack whose processor has the
63                least work load (greedy) */
64        {
65             Stop the execution of job k (preempt k),
66             Push the job T_i,j onto the stack on top of T_k,
67             Start executing T_i,j,
68             Calculate the fixed priority of T_i,j using its
69               Start time s_i,j,: d'_i,j(t) = β_i(s_i,j + w_i – r_i,j)
70             Add the execution time of T_i,j to the total
71               workload of that stack (∑ Ws_l ),
72             Recalculate total workload of the
73                Processor l:
74                W_l = ∑ Wp_l + ∑ Ws_l
75        }
```

Check for missed Deadlines

```
76        /* at each time instance t, if any of the running jobs
77           on top of the stacks has reached its break point:
78              (t > Br_i,j),  Br_i,j = min (p_i || s_i,j +2w_i)
79            remove the job from the stack and send
80            it to the processor Garbage Collection
81            otherwise, if not preempted, continue its
82            execution */
```

Benefit Gained by Completed Jobs

```
83         /* for every completed job T_i,j calculate b_i,j */
84        b_i,j = w_i. β_i( f_i,j )
85        }
```

Total Benefit Calculation

```
86        /* calculate the sum of all benefits gained,
87           B is initially set to zero*/
89        B = B + b_i,j
90   }
```

$d_{i,j}(t)$ – *variable priority* of job $T_{i,j}$ at time t, before scheduling ($t < s_{i,j}$): $d_{i,j}(t) = \beta_i (t + w_i - r_{i,j})$ (5)

$d'_{i,j}$ – *fixed priority* of job $T_{i,j}$, when it is scheduled and starts running: $d'_{i,j} = \beta_i (s_{i,j} + w_i - r_{i,j})$ (6)

2.4. Our Algorithm

In this system, the events are new job arrival, job completion, and reaching the break point of a job. The algorithm takes action when a new job arrives, a running job completes, or when a running job reaches its break point. When new jobs arrive they will be prioritized, and partitioned among the processors. The job on top of each stack is the job that is running and all other jobs in the stacks are preempted.

A. Prioritizing:

The priority of each ready and unscheduled job (located in each pool) at time t which is denoted by $d_{i,j}(t)$ (for $t \leq s_{i,j}$) is variable with time. However, when a job T_k (k can be any i,j) starts its execution, its priority is calculated as $d'_k = \beta_k (s_k + w_k - r_k)$ (lines 19 and 68 of the pseudo-code). The notation d'_k is used for the fixed priority of the running job T_k on top of the stack. This priority is given to the job T_k when it starts its execution. Its start time, s_k, is used in the function instead of variable t, therefore its priority is no longer dependent on time. Since s_k , w_k and r_k are all constants, the priority of a job will not change after its start time (for $t > s_k$).

B. Scheduling / Execution / Preemption:

Once a new job $T_{i,j}$ is released, if there is a processor such that its stack is empty (lines 11 through 22), then the newly released job is pushed onto the stack and starts running. If there is no idle processor, but for any running processor $d_{i,j}(t) > 4d'_k$ (lines 58 through 66), the job $T_{i,j}$ preempts the currently running one, and starts its execution. The analysis [9] shows that the factor 4 in the preemption condition ($d_{i,j}(t) > 4d'_k$) plays role in constant ratio competitiveness being equal to $10C^2$.

C. Online Partitioning (Load-Balancing/Greedy):

If more than one high priority job is able to preempt some running job(s), to decide which job should be sent to which stack, we send the largest job to the processor with the minimum remaining work load, the second largest job to the processor with the second smallest remaining work load, so on so forth. This way we are able to balance the work load among the processors.

However, in case there is only one high priority job at a time instance which can preempt more than one running job, we assign it to the stack of the processor with minimum remaining execution time (Greedy approximation). If the priority of the released job is not high enough to be scheduled right away, it will be partitioned among the pools of the processors using an online choice of load balancing or Greedy approximation (lines 39 through 75).

D. Reaching Break Point:

If a job reaches its break point and its execution is not completed yet, it will not be able to gain any benefit; therefore, it will be popped from the stack and sent to the garbage collection. The break point or deadline of a job is either its period or twice its execution time after it starts running, whichever is less.

E. Reward Accumulation / Completion / Discarding:

When a currently running job on a processor completes, it is popped from the stack. Then, the processor runs the next job on its stack (i.e. resumes the last preempted job) if $d_{i,j}(t) \leq 4d'_k$ for all the jobs $T_{i,j}$ in its pool. Otherwise, it gets the job with max $d_{i,j}(t)$ from its pool, pushes it onto the stack and runs it. The completed jobs or those that reach their break points are going to be sent to the garbage collection. If a job completes before reaching its break point, its gained benefit is calculated and added to the total benefit.

3. FUTURE WORK

Ongoing work conducts both theoretical and experimental performance analysis of this algorithm. In order to compare it with state-of-the-art, we consider metrics such as total gained reward, tardiness and overall response time. It also studies the upper bounds on task utilization.

REFERENCES

[1] A. Elnably, K. Du, P. Varman, "Reward scheduling for QoS in cloud applications," in Proc. of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012.

[2] J. Gubbi, R. Buyya, S. Marusic , M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems 29 (2013) 1645–1660

[3] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in Proc. of the 11th International Workshop on Parallel and Distributed Real-time Systems, April 2003.

[4] H. Alhussian, N. Zakaria, F. A. Hussin, "An efficient real-time multiprocessor scheduling algorithm," in Journal of Convergence Information Technology, January 2014.

[5] M. Amirijoo, J. Hansson, and S. H. Son, "Specification and management of QoS in real-time databases supporting imprecise computations," in IEEE Transactions on Computers, vol. 55, pp. 304–319, March 2006.

[6] H. Aydin, R. Melhem, D. Mosse and P. M. Alvarez, "Optimal reward-based scheduling for periodic real-time tasks," in IEEE Transactions on Computers, vol. 50, no. 2, February 2001.

[7] I-H. Hou, P.R. Kumar, "Scheduling periodic real-time tasks with heterogeneous reward requirements," in Proc. of the 32nd IEEE Real-Time Systems Symposium, 2011.

[8] B. Awerbuch, Y. Azar, and O. Regev, "Maximizing job benefits online," in Proc. of the 3rd International Workshop, APPROX, Germany, September 2000.

[9] B. Sanati and A.M.K. Cheng, "Maximizing job benefits on multiprocessor systems using a greedy algorithm," in WiP session of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April, 2008.

[10] B. Sanati and A.M.K. Cheng, "Efficient Online Benefit-Aware Multiprocessor Scheduling Using an Online Choice of Approximation Algorithms," in Proc. of the 11th IEEE International Conference on Embedded Software and Systems (ICESS 2014), Paris, France, August 20-22, 2014.

[11] J.H. Anderson, J.P. Erickson, U.C. Devi, B.N. Casses, "Optimal semi-partitioned scheduling in soft real-time systems," in Proc. of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), August 20-22, 2014.