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 overhead 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 was 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
In the execution phase, in case by defining their deadline, preemption would be the different jobs of any task get processed and recorded them in pool for the patients. It process and records them drastically reducing method is the possibility to delay jobs without no precedence constraints among them.

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 \( j = 1, 2, 3, \ldots \). 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:

\[
Br_{i,j} = \min (p_i \| s_{i,j} + 2w_i)
\]

- \( \beta_i(t) \) – 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 \( \beta_i(t) \):

\[
\frac{\beta_i(t)}{\beta_i(t + w_i)} \leq C
\]

**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 \( T_{i,j} \):

\[
f_{i,j} = c_{i,j} - r_{i,j}
\]

- \( b_{i,j} \) – benefit, gained by a completed job \( T_{i,j} \):

\[
b_{i,j} = w_i \beta_i(f_{i,j})
\]
**LBBA Algorithm for Periodic Tasks**

1. **Required:** One or more jobs arrive at time $t \geq 0$
2. 
   ```
   Job Arrival
   ```
3. /* TempList: list of ready jobs waiting for distribution among processors */
4. 
5. **Append** the arrived job(s) to the TempList
6. 

**Benefit-Based Scheduling**

7. **Calculate** the priority of each job $T_{ij}$ in the TempList:
8. $d_{ij}(t) = \beta_j(t + w_i + r_{ij})$
9. **Sort** TempList based on the priority
10. If (at least one stack is empty)
11. 
   ```
   Push
   ```
12. the highest priority job(s) $T_{ij}$ onto empty stack(s) of idle processor(s) $l$;
13. **Add** its execution time $w_i$ to total workload of the stack of the processor $l$ ($\sum W_p$),
14. **Recalculate** total workload of processor $l$:
15. $W_l = \sum W_p + \sum W_s$
16. **Calculate** the fixed priority of $j$ using its start time $s_{ij}$:
17. $d_{ij}(t) = \beta_j(s_{ij} + w_i - r_{ij})$
18. **Start** executing $j$,
19. 
20. Else
21. 
22. /* no stack is empty */
23. /* preempt if possible otherwise distribute among the pools */
24. 
25. **Compare** the priority of the ready jobs in TempList with the priority of the running jobs (indicated by index $k$) on top of the stacks:
26. If ($d_{ij}(t) \leq 4d_{ij}'$ for (each job $T_{ij}$ in TempList and each running job $T_{kl}$)
27. 
28. /* no preemption allowed */
29. /* partition the ready jobs among pools of the processors */
30. 
31. **For** (each job $T_{ij}$ in TempList)
32. 
33. **Sort** the processors in ascending order of their total remaining workload on their pools and stacks:
34. $W_l = \sum W_p + \sum W_s$
35. **Append** the job $T_{ij}$ with largest execution time $w_i$ to the pool of the processor $l$ with minimum remaining work load; /* load balancing */
36. **Remove** $T_{ij}$ from TempList;
37. **Add** its execution time $w_i$ to total workload of the pool of processor $l$ ($\sum W_p$);
38. **Recalculate** total workload of processor $l$:
39. $W_l = \sum W_p + \sum W_s$
40. 
41. 

**Greedy Approximation (multiple-choice Preemption)**

42. /* If $T_{ij}$ has more than one choice of processors, it will be pushed onto the stack whose processor has the least work load (greedy) */
43. 
44. 
45. **Else**
46. /* if ($d_{ij}(t) > 4d_{ij}'$) then ($T_{ij}$ preempts $T_{kl}$)
47. **Check for missed Deadlines**
48. **Benefit Gained by Completed Jobs**
49. **Total Benefit Calculation**

---

47. 
48. processor $l$ with minimum remaining work load; /* load balancing */
49. 
50. **Remove** $T_{ij}$ from TempList;
51. **Add** its execution time $w_i$ to total workload of the pool of processor $l$ ($\sum W_p$);
52. **Recalculate** total workload of processor $l$:
53. $W_l = \sum W_p + \sum W_s$
54. 
55. 
56. 
57. 
58. 
59. /* if ($d_{ij}(t) > 4d_{ij}'$) then ($T_{ij}$ preempts $T_{kl}$)
60. **Greedy Approximation (multiple-choice Preemption)**
61. **Greedy Approximation (multiple-choice Preemption)**
62. **Greedy Approximation (multiple-choice Preemption)**
63. **Greedy Approximation (multiple-choice Preemption)**
64. **Greedy Approximation (multiple-choice Preemption)**
65. **Greedy Approximation (multiple-choice Preemption)**
66. **Greedy Approximation (multiple-choice Preemption)**
67. **Greedy Approximation (multiple-choice Preemption)**
68. **Greedy Approximation (multiple-choice Preemption)**
69. **Greedy Approximation (multiple-choice Preemption)**
70. **Greedy Approximation (multiple-choice Preemption)**
71. **Greedy Approximation (multiple-choice Preemption)**
72. **Greedy Approximation (multiple-choice Preemption)**
73. **Greedy Approximation (multiple-choice Preemption)**
74. **Greedy Approximation (multiple-choice Preemption)**
75. **Greedy Approximation (multiple-choice Preemption)**
76. **Greedy Approximation (multiple-choice Preemption)**
77. **Greedy Approximation (multiple-choice Preemption)**
78. **Greedy Approximation (multiple-choice Preemption)**
79. **Greedy Approximation (multiple-choice Preemption)**
80. **Greedy Approximation (multiple-choice Preemption)**
81. **Greedy Approximation (multiple-choice Preemption)**
82. **Greedy Approximation (multiple-choice Preemption)**
83. **Greedy Approximation (multiple-choice Preemption)**
84. **Greedy Approximation (multiple-choice Preemption)**
85. **Greedy Approximation (multiple-choice Preemption)**
86. **Greedy Approximation (multiple-choice Preemption)**
87. **Greedy Approximation (multiple-choice Preemption)**
88. **Greedy Approximation (multiple-choice Preemption)**
89. **Greedy Approximation (multiple-choice Preemption)**
90. **Greedy Approximation (multiple-choice Preemption)**
\( d_{i,j}(t) \) – variable priority of job \( T_{i,j} \) at time \( t \), before scheduling \((t < s_{i,j})\): \( d_{i,j}(t) = \beta_{i}(t + w_{i} - r_{i}) \) \( (5) \)

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

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 \( t > s_{i,j} \)).

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 and 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