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

Behnaz Sanati, Albert M. K. Cheng, Nicholas Troutman

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-16-02
May 27, 2016

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

Abstract

In this paper, we propose a novel semi-partitioning approach with an online choice of two approximation algorithms, Greedy and Load-Balancing, to enhance scheduling of soft real-time periodic tasks in homogeneous multiprocessor systems. Our objective is to optimize the QoS by minimizing the deadline misses and maximizing the total benefit (or reward) obtained by completed tasks. We analyze our model for the systems with implicit and non-implicit deadlines and evaluate them through extensive experiments. 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 and medical monitoring systems.

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

Behnaz Sanati, Albert M. K. Cheng, Nicholas Troutman

Abstract

In this paper, we propose a novel semi-partitioning approach with an online choice of two approximation algorithms, Greedy and Load-Balancing, to enhance scheduling of soft real-time periodic tasks in homogeneous multiprocessor systems. Our objective is to optimize the QoS by minimizing the deadline misses and maximizing the total benefit (or reward) obtained by completed tasks. We analyze our model for the systems with implicit and non-implicit deadlines and evaluate them through extensive experiments. 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 and medical monitoring systems.

Index Terms

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

I. Introduction

A. Background

Multiprocessor systems are widely used in a fast-growing number of real-time applications as well as 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 (SRT) systems, missing few deadlines does not drastically affect the system performance. However, it would compromise the quality of the service (QoS).

In such systems, jobs meeting their deadlines will g[1]ain a *benefit* (also called *reward*) for the system. Hence, researchers focus on maximizing benefits to improve the QoS. Besides the total benefit, other factors also influence QoS, such as overall response time (makespan plus scheduling time) and deadline-miss ratio also called *tardiness*. 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].

B. Related Works

The above issues are discussed at some length by Srinivasan *et al.* [3]. Elnably *et al.* [1] study fair resource allocation and propose a benefit-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 the number of preemptions and migrations.

Amirijoo, Hansson and Son [5] have discussed specification and management of QoS in real-time databases supporting imprecise computations. Benefit-based scheduling of periodic tasks has also been studied by Aydin *et al.* [6], and Hou and Kumar [7]. Awerbuck *et al.* [8] proposed a benefit-maximizing model for scheduling aperiodic tasks on uniprocessor systems which can also be applied to multiprocessors. We have also previously studied benefit-based scheduling of aperiodic real-time tasks on multi-processor systems. We proposed two algorithms,

[1] Supported in part by the National Science Foundation under Awards No. 0720856 and No. 1219082

GBBA [9] and LBBA [10], and provided performance analysis and comparative experimental results of those algorithms versus another state-of-the art algorithm [8].

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. In EDF-os, the number of processors to which jobs of a migrating task can migrate to, are limited to two, and also each processor can be assigned to only two migrating tasks. The goal in these EDF-based semi-partitioning strategies is to minimize tardiness.

C. Our Objective

Our objective in this study is to enhance the QoS by minimizing missed deadlines and maximizing the total benefit obtained by completed periodic tasks. Hence, we allow different jobs of any task to be assigned to different processors (migration at job boundaries) based on their benefit-based priorities and workload of the processors. This method can also be used as a framework to direct SRT systems with mixed set of tasks (aperiodic and periodic) by defining their deadlines accordingly.

D. Our Contribution

We previously proposed the LBBA method for scheduling aperiodic tasks [12], which achieved significant improvements in reducing the overall response time (i.e., scheduling time plus makespan of the task sets), maximizing the total benefit and minimizing missed deadlines, all of which enhance QoS. However, that method is designed for scheduling one instance (i.e., aperiodic) tasks, and cannot be used for solving the problem of scheduling periodic soft real-time tasks on multi-processor systems, on which relatively very little research has been done.

In this work, we propose a new online benefit-based semi-partitioning approach to schedule periodic soft real-time tasks in homogeneous multiprocessor systems. Scheduling is based on the task priority, depending on the benefit density function of each task. We apply an online choice of two approximation algorithms (Load-Balancing and Greedy approximation) for partitioning lower priority tasks that are waiting, at job boundaries. No migration is allowed after a job (or sub-task) is assigned to a processor. This technique provides:

- An optimized usage of the processing time by approximately balancing the workload of the processors, which reduces the idle times and makespan
- Reduces the NP-hard problem of multiprocessor real-time scheduling to uniprocessor scheduling
- When different benefit density functions are assigned to different tasks in a system, it maximizes the total gained benefit by prioritizing tasks based on their benefit density functions.
- This method, has advantages over existing semi-partitioning schedulers, such as:
    - No prior information is needed for scheduling. Hence, unlike other semi-partitioning methods, there is no offline phase.
    - In EDF-os, the number of processors on which different jobs of each task can be processed in limited to two, and also the each processor cannot accept jobs from more than two migrating tasks. There is neither of such limitations in our proposed method.
    - It reduces overhead by not allowing migration in the middle of job executions.

In the next section, we explain our novel semi-partitioning hybrid model, which combines benefit and cost models, for optimizing quality of service in soft real-time systems. In section III, we provide the theoretical analysis of this algorithm (LBBA for periodic tasks) and propose two more variations of LBBA, one with different deadline definition and the other one with a different factor considered for load-balancing. Section IV includes the performance analysis of all three proposed approaches based on the results of our extensive simulation experiments on synthetic task sets. Section V, concludes the advantages of this work and suggests the future work.

II. LBBA For Periodic Tasks

In this section, we define the system and task model, methodology and notations/phrases used in our proposed LBBA algorithm for periodic tasks.

A. *System and Task Model*

A multiprocessor system with *m* identical processors is considered for semi-partitioned, preemptive scheduling of periodic soft real-time task sets with implicit deadline (or non-implicit, depending on the application). 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 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.

B. *Methodology*

We propose a hybrid model (combining benefit and cost models) for online scheduling of periodic tasks in SRT systems. In this method, we apply our novel partitioning technique, in addition to online choice of approximation algorithms as follows.

1) *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 instead of using a shared pool, each processor has its own pool for the ready tasks assigned to it. Partitioning jobs at their release time reduces the NP-hard problem of multiprocessor scheduling to multiple cases of uniprocessor scheduling.

2) *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. We proposed this approximately balanced partitioning method in earlier phases of this research [10] as a part of our proposed scheduling method, LBBA (for aperiodic tasks), and showed some advantages of applying this technique in [10] and [12], such as CPU usage optimization by reducing idle times and makespan.

Figure 1, summarizes our methodology for LBBA scheduling of periodic soft real-time task sets on multiprocessor systems.

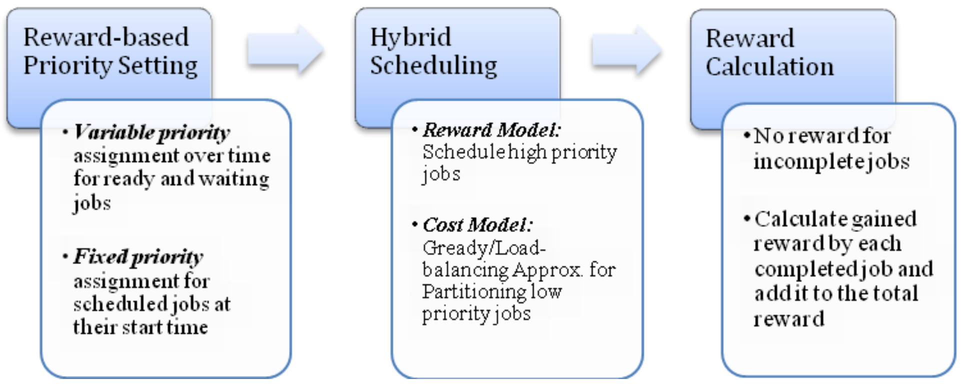
Reward-based Priority Setting
• Variable priority assignment over time for ready and waiting jobs
• Fixed priority assignment for scheduled jobs at their start time
Hybrid Scheduling
• Reward Model: Schedule high priority jobs
• Cost Model: Gready/Load-balancing Approx. for Partitioning low priority jobs
Reward Calculation
• No reward for incomplete jobs
• Calculate gained reward by each completed job and add it to the total reward

Fig. 1. Our Methodology

C. Definitions

We provide the definitions of the phrases and notations used in this paper as follows:

1) 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 worst-case execution time w_i as well as its period p_i. An instance of a periodic task T_i (*i.e* release) is known as a job and is denoted as Ti,j, where $j = 1, 2, 3, \ldots$. The *implicit deadline* of a job is the arrival time of its successor. For example, the deadline of the j^{th} job of Ti, which is $T_{i,j}$, would be the arrival time of job $Ti,(j+1)$, that is at jp_i. However, it can be *non-implicit* and defined based on objectives and criticalities of the systems and applications.

2) Task Utilization:

Another important parameter used to describe a task Ti is its utilization and is defined as $u_i = w_i / p_i$. The utilization of a task is the portion of time that it needs to execute after it has been released and before it reaches its deadline.

3) Notations:

We define the notations used throughout this paper as follows:

p_i – *period of task* T_i
w_i – *worst-case execution time* of *task* T_i, considered as *workload* of *task* T_i in this paper
$r_{i,j}$ – *release time* of job Ti, j
$s_{i,j}$ – *start time* of job Ti, j
$c_{i,j}$ – *completion time* of job $Ti,,j$
$Br_{i,j}$ – *benefit-based break point* of job $Ti,,j$, is:

$$Br_{i,j} = s_{i,j} + 2w_i \tag{1}$$

This means if twice the execution time of a running job has passed from its start time and it has not finished its execution yet, then it cannot gain any benefit for the system.

$\beta_i(t)$ – *benefit density function* of task Ti 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 Ti,j, since it has certainly not completed its execution at time t. The above condition guaranties that in case a job is delayed as long as its worst-case execution time, then its gained benefit decreases at most by the constant C.

$f_{i,j}$ – *flow time* of job Ti,j:

$$f_{i,j} = c_{i,j} - r_{i,j} \tag{2}$$

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

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

Since β_i is a non-negative, non-increasing function, the sooner a job finishes the more benefit it gains. Also, between two jobs with the same benefit density function and same flow time, the one with larger execution time adds more benefit to the system.

$d_{i,j}(t)$ – *variable priority* of job Ti,j at time t, before scheduling ($t < s_{i,j}$):

$$d_{i,j}(t) = \beta_i (t + w_i - r_{i,j})$$

$d'_{i,j}$ – *fixed priority* of job Ti,j, when it is scheduled and starts running:

$$d'_{i,j} = \beta_i (s_{i,j} + w_i - r_{i,j})$$

$D_{i,j}$ – *deadline* of job *Ti,,j,*

Note: We propose and analyze our model with two different types of deadlines, implicit and non-implicit.

Implicit deadline (Next- Job- Release time):

$$D_{i,j} = r_{i,(j+1)}$$
$$D_{i,j} = r_{i,j} + p_i \tag{3}$$

Non-implicit deadline (Next-Job-Completion time):

$$D_{i,j} = c_{i,(j+1)}$$

U – maximum possible utilization of the system with *m* identical processors:

$$U = m$$

u_i – *utilization* of every job of the task *Ti* :

$$u_i = w_i / p_i$$

$£_i$ – *laxity* of job *Ti,j* :

$$£_i = p_i - w_i \tag{4}$$

$\delta_{i,j}$ – *delay* of job *Ti,j* , that is the time *Ti,j* has to wait after it is released until it is scheduled and starts its execution:

$$\delta_{i,j} = s_{i,j} - r_{i,j} \tag{5}$$

$\varphi_{i,j}(k)$ – *time elapsed during the* k^{th} *preemption* of *Ti,j*

D. Our Algorithm

In this system, the tasks are periodic and the events are new job (or sub-task) 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, then either scheduled and start running on the assigned processors or partitioned and sent to the pools of the processors. The job on top of each stack is the job that is running and all other jobs in the stacks are preempted. The jobs on the stacks or the ones in the pools cannot migrate to any other processor. However, different jobs of a task can be assigned to different processors at their arrival time. We call this algorithm *LBBA-bid*, that is *LBBA with **b**enefit-aware **i**mplicit **d**eadlines.*

The summary of the algorithm is provided in pseudo-codes on the next page and consists of the following phases:

1) 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 $t > s_k$).

2) 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. This preemption condition ($d_{i,j}(t) > 4d'_k$) not only plays role in constant ratio competitiveness being equal to $10C^2$ [8], but also limits the number of preemptions and the overhead caused by them. This is provided by preventing a new job $T_{i,j}$ from preempting the running jobs with lower priorities unless its priority is at least four times higher than theirs.

LBBA Algorithm for Periodic Tasks

```
Required: One or more jobs arrive at time $t \geq 0$
{
```

Job Arrival

```
    /* TempList: list of ready jobs waiting for distribution among processors */

    Append the arrived job(s) to the TempList
```

Benefit-Based Scheduling

```
    Calculate the priority of each job $j$ in the TempList:
        $d_j(t) = B_j(t + w_j - r_j)$
    Sort TempList based on the priority
    If (at least one stack is empty)
    {
        Push the highest priority job(s) $j$ onto empty stack(s) of idle processor(s) $i$;
        Add its execution time $w_j$ to total workload of the stack of the processor $i$ ($\sum W_{si}$),
        Recalculate total workload of processor $i$:
            $W_i = \sum Wp_i + \sum Ws_i$
        Calculate the fixed priority of $j$ using its start time $s_j$:
            $d'_j(t) = B_j(s_j + w_j - r_j)$
        Start executing $j$,
    }
    Else
    {
        /* no stack is empty */
        /* preempt if possible otherwise distribute among the pools */
        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:
        If ($d_j(t) \leq 4d'_k$ for ( each job $j$ in TempList and each running job $k$) )
        {
            /* no preemption allowed */
            /* partition the ready jobs among pools of the processors */
```

Load-Balancing Approximation (for Partitioning)

```
            For (each job $j$ in TempList)
            {
                Sort the processors in ascending order of their total remaining workload on their pools and stacks :
                    $W_i = \sum Wp_i + \sum Ws_i$
                Append the job $j$ with largest execution time $w_j$ to the pool of the processor $i$ with minimum remaining work load;   /* load balancing */
                Remove $j$ from TempList;
                Add its execution time $w_j$ to total workload of the pool of processor $i$ ($\sum Wp_i$);
                Recalculate total workload of processor $i$:
                    $W_i = \sum Wp_i + \sum Ws_i$
            }
        }
        Else
            /* if ($d_j(t) > 4d'_k$) then ( $j$ preempts $k$)*/
```

Greedy Approximation (multiple-choice Preemption)

```
            /* If $j$ has more than one choice of processors, it will be pushed onto the stack whose processor has the least work load (greedy) */
        {
            Stop the execution of job $k$ (preempt $k$),
            Push the job $j$ onto the stack on top of $k$,
            Start executing $j$,
            Calculate the fixed priority of $j$ using its Start time $s_j$: $d'_j(t) = B_j(s_j + w_j - r_j)$
            Add the execution time of $j$ to the total workload of that stack ($\sum Ws_i$ ),
            Recalculate total workload of the Processor $i$:
                $W_i = \sum Wp_i + \sum Ws_i$
        }
```

*Check for missed Deadlines (**Benefit-aware/Implicit**)*

```
        /* at each time instance $t$, if any of the running jobs on top of the stacks have reached its break point:    $t > \min(D_{i,j}, Br_{i,j})$,  $Br_{i,j} = s_{i,j} + 2w_i$
           remove the job from the stack and send it to the processor Garbage Collection;
           otherwise, if not preempted, continue its execution */
```

Benefit Gained by Completed Jobs

```
        /* for every completed job $j$ calculate $b_j$ */
        $b_j = w_j \cdot \beta_j(f_j)$
    }
```

Total Benefit Calculation

```
    /* calculate the sum of all benefits gained, $q$ being the number of completed jobs */
    $B = \sum_{j=1}^{q} bj$
}
```

3) 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).

4) Reaching Break Point or Deadline:

If a job reaches its break point or deadline (line 78) 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 deadline of a job is its period (Next-Job-Release time of the same task) and its break point is twice its execution time after it starts running. A job must finish its execution before its deadline or break point (whichever is less) to be considered as completed (lines 76-82).

5) Completion / Discarding / Benefit Calculation:

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 *Ti,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, its gained benefit is calculated and added to the total benefit (lines 83 through 90).

III. Analysis

A. LBBA-bid Analysis

In LBBA-bid, a job must complete by the end of period, i.e., before the next job of the same task is released. The benefit-awareness attribute of LBBA also requires a job not to take longer than twice its worst-case execution time after its start time to complete; otherwise, it would be discarded from the system without gaining any benefit. This restriction will induce an upper bound on the delay each job may have after release till it is scheduled and starts its execution. From the definition of break point (eq. (1)),

$$Br_{i,j} = s_{i,j} + 2w_i$$

If $Br_{i,j} > D_{i,j}$, then $T_{i,j}$ can continue until the next job arrives, and if $T_{i,j}$ is not preempted while running, the following condition must hold for it to meet its deadline:

$$s_{i,j} + w_i \leq D_{i,j}$$

From eq.(3):

$$s_{i,j} + w_i \leq r_{i,j} + p_i$$
$$s_{i,j} - r_{i,j} \leq p_i - w_i$$

From (4) and (5):

$$\boldsymbol{\delta}_{i,j} \leq £_i$$

Therefore, the maximum delay in starting a job execution is equal to its laxity. This defines the ***upper bound on the start time*** as follows:

$$Max\ (s_{i,j}) = r_{i,j} + £_i$$

Schedulability Condition - If this occurs to a job, then it cannot be preempted during its execution, to be able to meet its deadline. If a higher priority job is scheduled on the same processor and preempts it, then it will miss the deadline and will not gain any benefit.

Corollary 3.1.1 – If a job $T_{i,j}$ is scheduled at its $Max\ (s_{i,j})$, then in order to meet its deadline it should not get preempted.

Theorem – *If the utilization of a job $T_{i,j}$ is equal to or more than half ($w_i \geq \frac{1}{2} p_i$) and ($Br_{i,j} \leq D_{i,j}$), then it has to start running as soon as it is released, without preemption, to be able to meet its deadline.*

If $Br_{i,j} \leq D_{i,j}$, then

$$s_{i,j} + 2w_i \leq D_{i,j}$$
$$s_{i,j} + 2w_i \leq r_{i,j} + p_i$$
$$s_{i,j} - r_{i,j} \leq p_j - 2w_i$$
$$\boldsymbol{\delta}_{i,j} \leq p_i - 2w_i \qquad (6)$$

If $u_i \geq \frac{1}{2}$,

$$p_i \leq 2w_i$$

, and from (6):

$$Max\ (\boldsymbol{\delta}_{i,j}) = 0$$

So, the theorem is proved. ■

On the other hand, in order to gain any benefit, the following condition must hold:

$$f_{i,j} \leq p_i$$

By definition, eq. (2):

$$c_{i,j} - r_{i,j} \leq p_i$$

Assume $T_{i,j}$ has been preempted k times, then the total time elapsed during preemptions of $T_{i,j}$ is denoted by $\Sigma^k \boldsymbol{\varphi}_{i,j}$. Therefore,

$$w_i + \Sigma^k \boldsymbol{\varphi}_{i,j} + \boldsymbol{\delta}_{i,j} \leq p_i \qquad (7)$$

Corollary 3.1.2 - The **upper bound on preemption time** is the laxity of the job $T_{i,j}$, and that is when it starts at release time without any delay (from eq. (7)).

$$Max\ (\Sigma^k \boldsymbol{\varphi}_{i,j}) = £_i$$

This condition holds for the highest priority jobs which can preempt another job at their release time, or get immediately scheduled on an idle processor. Jobs that are partitioned into the pools and have a waiting time (delay) cannot have preemption time up to their laxities; otherwise, they would miss their deadline.

Hence, there would be cases of missed deadlines if the delay in scheduling and/or total time a job spends in preemptions would pass the upper bounds or the above conditions are violated. We offer the following propositions in order to allow more jobs to complete without compromising the QoS.

B. Propositions

We explain two propositions to modify deadline and load-balancing factors and study their effects on the performance of the system.

1. Non-Implicit Deadline (LBBA-bnc)

Proposition 1 – In order to let more jobs continue their execution until they complete and gain some benefit for the system, we relax the benefit aware implicit deadline (bid) by changing to ***b**enefit-aware **n**on-implicit deadline of* ***n**ext-job-**c**ompletion time (bnc):*

$$D_{i,j} = c_{i,(j+1)}$$

This can be done if the applications' expectation of job completion allows this relaxation of deadline. Then, if $T_{i,j}$ is running on one processor and before its completion $T_{i,j+1}$ is released, there would be two possible cases. One is that the priority of $T_{i,j+1}$ is not high enough and it has to be partitioned and sent to a pool. If it is not on the same processor of $T_{i,j}$, and $T_{i,j}$ completes while $T_{i,j+1}$ is waiting or it has started and still running, the benefit gained by $T_{i,j}$ is added to the total benefit and its processing time has not been wasted. Also, if $T_{i,j+1}$ is waiting on the pool of the same processor $T_{i,j}$ is running on, then it has to complete before the laxity of $T_{i,j+1}$ ends. In this case, both jobs meet their deadlines.

We will evaluate LBBA-bid and LBBA-bnc by comparing their total benefits and the number of completed jobs through experiments.

2. Utilization Balancing (UBBA)

Many EDF-based algorithms consider the utilization of the tasks (*u*) instead of the workload or execution time (*w*), with the objective of making the task sets schedulable and reducing tardiness. LBBA is balancing the workload among processors. Therefore, to be able to study the difference in the performance, we propose another version of our algorithm which balances the utilizations among the processors.

Proposition 2 – In the *UBBA* (***u****tilization-****b****alanced* ***b****enefit-****a****ware*) algorithm, we replace the load-balancing part with the following:

Utilization-Balancing Approximation (for Partitioning)

```
For (each job j in TempList)
{
  Sort the processors in ascending order of
    their total remaining utilization on their
    pools and stacks :
      $U_i = \sum Up_i + \sum Us_i$     // i is proc. index
   Append the job j with largest
      utilization time $u_j$ to the pool…
```

The same method applies to the greedy approximation, and also every time a job is added to a pool or pushed on a stack its utilization will be added to the total remaining utilization of that processor (instead of *w*).

C. *An Example*

We demonstrate how the proposed algorithms schedule a set of tasks through an example. Assume a system with 2 identical processors and three periodic tasks as shown in the Table 1.

TABLE 1
AN EXAMPLE OF 3 PERIODIC TASKS

	T_1	T_2	T_3
W	3	2	7
P	5	3	10

The LCM of their periods is 30. Therefore, we illustrate the scheduling processes within the first 30 units of time. During this time period, 6 instances of T_1, 10 instances of T_2 and 3 instances of T_3 will be released. Their total utilizations will be 59/30 (18/30 + 20/30 + 21/30). This is less than 2, i.e., the maximum possible utilization of a 2 processor system. Therefore, the necessary condition for the task set to be schedulable is met, although it is not sufficient. Assuming that the tasks are synchronous and released at time t = 0, with the same benefit density function (e.g., $f(x) = 1/x$), the LBBA-bid scheduling process is as follows:

The priority of each task is calculated and the tasks are sorted in a descending order of their priorities. $T_{2,1}$ (the first instance of T_1) with the highest priority is pushed on the stack of processor 1, denoted as P_1, $T_{1,1}$ with the second highest priority is scheduled on P_2 and $T_{3,1}$ with the lowest priority has to wait. Since the current workload on P_1 is 2 and on P_2 is 3, $T_{3,1}$ is partitioned and sent to the pool of P_1, with the lowest current workload or execution time.

At time $t = 2$, $T_{2,1}$ is completed and its benefit is calculated and equals 1 for the given benefit density function. Then, $T_{3,1}$ is transferred from the pool to the stack of P_1 and starts running. $T_{1,1}$ is completed at $t = 3$, the same time that the next instance of T_2 (denoted as $T_{2,2}$) is released and having P_2 idle, it starts running on P_2 immediately. The benefit of $T_{1,1}$ is calculated and added to the total benefit. The chronological status of the system is listed below:

$t = 5$:

$T_{2,2}$ finishes, $T_{1,2}$ is released and starts on P_2.
Total benefit = 3

t = 6:

$T_{2,3}$ is released; its priority is set to 1/2 (1/ (6+2-6)), compared to the priority of the running jobs, 1/3 for $T_{1,2}$, and 1/9 for $T_{3,1}$, $T_{2,3}$ preempts $T_{3,1}$ (1/2 > 4/9) and starts on P_1.

t = 8:

$T_{1,2}$ finishes on P_2; $T_{2,3}$ finishes on P_1; and their benefits are calculated and added to the total benefit. Total benefit =5

$T_{3,1}$ resumes on P_1.

t = 9:

$T_{2,4}$ is released and starts on P_2.

t = 10:

$T_{1,3}$ and $T_{3,2}$ are released; No preemption is possible. Current remaining workload of each processor is as follows:

W_1 = 1 (remained from $T_{3,1}$)

W_2 = 1 (remained from $T_{2,4}$)

So, the scheduler sends $T_{1,3}$ to the pool of P_1 and $T_{3,2}$ to the pool of P_2.

t = 11:

$T_{3,1}$ and $T_{2,4}$ finish. $T_{1,3}$ and $T_{3,2}$ are transferred from the pools to the stacks of P_1 and P_2 respectively, and start. The benefits of $T_{3,1}$ (w_{31}/f_{31} = 7/11) and $T_{2,4}$ (2/2 = 1) are added to the total benefit resulting in 6.64.

t = 12:

$T_{2,5}$ is released. Its priority is not high enough to preempt any of $T_{1,3}$ and $T_{1,2}$. It is sent to the pool of P_1 with the least current workload:

W_1 = 2 (remained from $T_{1,3}$)

W_2 = 6 (remained from $T_{3,2}$)

t = 14:

$T_{1,3}$ finishes on P_1. Its benefit (3/4) is added to the total benefit resulting in 7.39. $T_{2,5}$ starts on P_1.

t = 15:

$T_{1,4}$ and $T_{2,6}$ are released. $T_{2,5}$ is incomplete and hence misses the deadline and gains no benefit. $T_{2,6}$ starts on P_1 after $T_{2,5}$ is sent to garbage collection. $T_{1,4}$ is sent to the pool of P_1, because:

W_1 = 2 (after starting $T_{2,6}$)

W_2 = 3 (remained from $T_{3,2}$)

t = 17:

$T_{2,6}$ finishes and $T_{1,4}$ starts on P_1.

Total benefit = 8.39

t = 18:

$T_{3,2}$ finishes on P_2. Its benefit is 7/8.

Total benefit = 9.265

$T_{2,7}$ is released and starts on P_2.

t = 20:

$T_{1,4}$ and $T_{2,7}$ finish.

Their benefits are 3/5 and 2/2 respectively.

Total benefit = 10.865

$T_{1,5}$ and $T_{3,3}$ are released and start on P_1 and P_2, respectively.

t = 21:

$T_{2,8}$ is released. It cannot preempt any jobs.

$T_{2,8}$ is sent to the pool of P_1 (W_1 = 2 vs. W_2 = 6)

t = 23:

$T_{1,5}$ ends. Its benefit is 1 (3/3).

Total benefit = 11.865

$T_{2,8}$ starts on P_1.

t = 24:

$T_{2,9}$ is released.

$T_{2,8}$ is incomplete, and misses its deadline.

$T_{2,9}$ replaces $T_{2,8}$ and starts on P_1.

t = 25:

$T_{1,6}$ is released.
W_1 = 1 (remained from $T_{2,9}$)
W_2 = 2 (remained from $T_{3,3}$)
$T_{1,6}$ is sent to the pool of P_1.

t = 26:
$T_{2,9}$ finishes on P_1. Its benefit is 1 (2/2).
Total benefit = 12.865
$T_{1,6}$ starts on P_1.

t = 27:
$T_{3,3}$ ends on P_2. Its benefit is 1 (7/7).
Total benefit = 13.865
$T_{2,10}$ is released and starts on P_2.

t = 29:
$T_{1,6}$ ends on P_1. Its benefit is 0.75 (3/4).
$T_{2,10}$ ends on P_2. Its benefit is 1 (2/2).
Total benefit = 13.865 + 0.75 +1 = 15.615

Now, we use LBBA-bnc for scheduling the same set of tasks on 2 identical processors. The priority settings, scheduling higher priority jobs and partitioning the rest of the ready jobs among the pools of the processors are the same as LBBA-bid, except for the case of having a job released when the
Previous job of the same task is not completed, yet. The example of such scenario is at *t* = 15, when $T_{2,5}$ is still running and $T_{2,6}$ is released; and at *t* = 24, when $T_{2,9}$ is released and $T_{2,8}$ is incomplete.

In LBBA-bnc, a job misses its non-implicit deadline if the next job of the same task completes (on another processor). Therefore, it allows two consecutive jobs of a task to have their executions, on two different processors, partially overlapped. Therefore, if the job that is released first completes first, it meets the deadline and can add its gained benefit to the total benefit. Hence, this relaxation of the deadline would reduce the tardiness (i.e., the number of missed deadlines) as shown in Figure 2.

At *t* = 15:
$T_{1,4}$ and $T_{2,6}$ are released. $T_{2,5}$ continues. $T_{2,6}$ and $T_{1,4}$ are sent to the pools of P_2 and P_1, respectively, in order to balance the workload.
W_1 = 1, W_2 = 3 (before partitioning)
W_1 = 4, W_2 = *5* (after partitioning)

The rest of the scheduling process is illustrated in Figure 2, along with the schedules provided by LBBA-bid and UBBA.

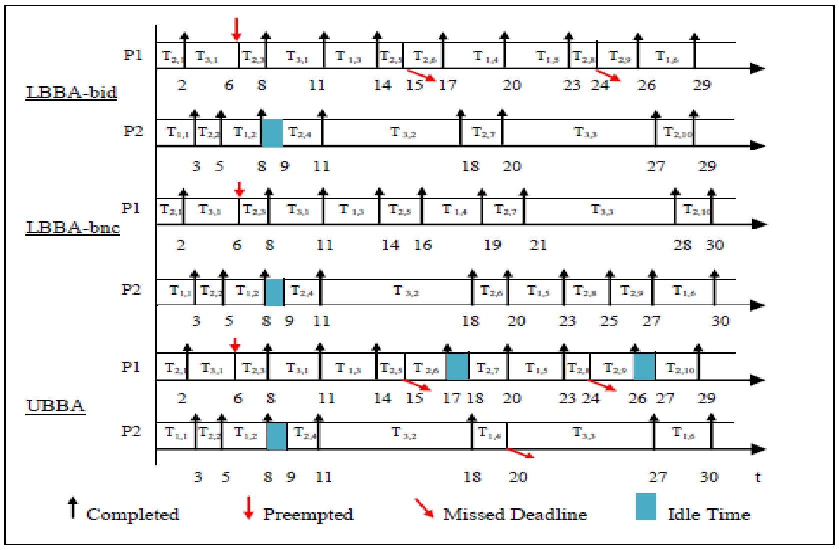

Fig. 2. Scheduling Diagrams

The UBBA will act the same at $t = 12$, for balancing the load based on utilizations, because the utilization of P_1 (2/3) is less than P_2 (6/7). However, its scheduling is different from LBBA-bid at $t = 15$, since utilization of P_1 (2/2) is more than P_2 (3/7). Therefore, $T_{1,4}$ will be sent to the pool of P_2.

As shown in the Figure, LBBA-bnc schedules all the jobs in this example, while two jobs in LBBA-bid and three jobs in UBBA method miss their deadlines.

IV. Experimental Evaluation

Through extensive experiments on synthetic periodic task sets, we conduct comparative performance evaluation for the three proposed algorithms.

- LBBA-bid (with **b**enefit-aware-**i**mplicit **d**eadline),
- LBBA-bnc (with **b**enefit-aware-**n**ext-job-**c**ompletion time deadline),
- UBBA (**U**tilization-**B**alanced **B**enefit-**A**ware)

Benefit maximization in LBBA has been proved for aperiodic tasks by theoretical and experimental analysis in [12]. Therefore, comparing with other state-of-the-art benefit-aware algorithms is not in the scope of this paper. We compare the schedulability (job completion rate) and the number of preemptions in the proposed algorithms with Global EDF, which is known as an optimal method for scheduling periodic tasks, to show how close these benefit-based scheduling methods are to the optimal solution, in term of schedulability, while maximizing the total benefit.

A. Performance Metrics

In this research, we considered the following measurements to evaluate and compare the performance of the three proposed algorithms:

- Total benefit gained by completed jobs
- Schedulability or Job Completion rate
- Number of preemptions

B. Experimental Setting

We implemented the algorithms using Netbeans 8.0.2, on Intel core i7- 2630QM CPU at 2 Ghz speed, 64 bit OS, 8 GB RAM and 6 MB cache. We randomly generated periodic task sets with uniform distribution of periods in the range of [1, 30] for 2, 4, 6, and 8 processors. The task utilizations were generated with uniform distribution as follows:

- 30% with light utilization in range of [0.001, 0.1]
- 40% medium utilization within [0.1, 0.4]
- 30% heavy utilization within [0.5, 0.9]

We generated the tasks until the total utilization passed the number of processors (100%), and then discarded the last generated task. We ran hundreds of trials for each multiprocessor setting and calculated the average amount of recorded results for the metrics.

C. Results and Discussion

The results of our extensive experiments are shown in the following graphs. Figure 3 implies that the total benefits gained by all three algorithms are very close so that they are shown as one line (on top of each other) in the graph. Please note that the benefits are in millions due to the very large LCM on 6 and 8 processors, which caused millions of jobs to be scheduled.

We did not include Global EDF in this graph, since it is not a benefit-aware algorithm and the gained benefit is not applicable to it.

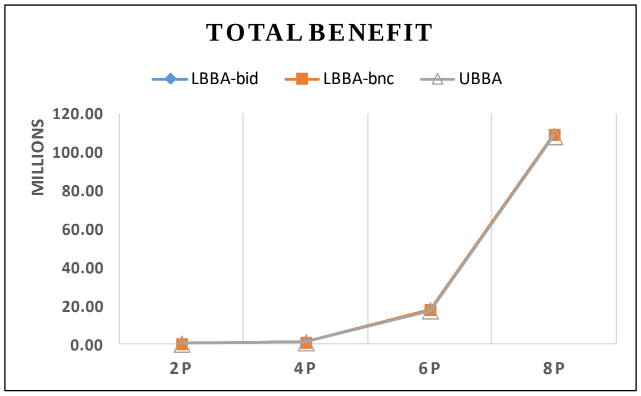

Fig. 3. Total benefit gained on average by LBBA-bid, LBBA-bnc and UBBA for 2, 4, 6, and 8 processors

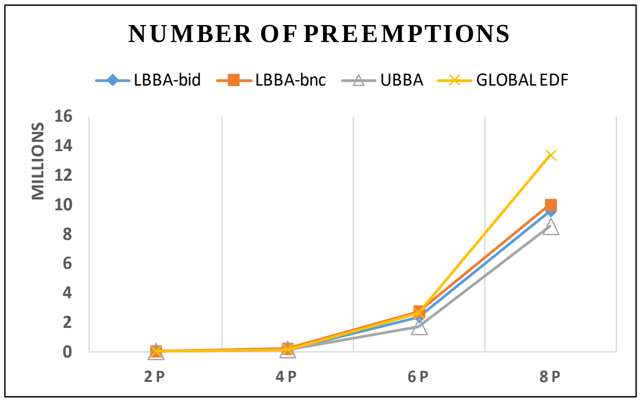

Fig. 4. Total number of preemptions on average by LBBA-bid, LBBA-bnc, UBBA and Global EDF, for 2, 4, 6, and 8 processors

Figure 4 compares the total number of preemptions on average, caused by each algorithm including Global EDF. The result showed that UBBA has the lowest number of preemptions during scheduling. The difference in their performance is more obvious when the number of processors increases. Since the system utilization in all cases is very close to 100%, there is a sharp increase in the number of jobs to be scheduled, with more processors in the system. Also, Global EDF had at least 30% more preemptions than our methods for 8 processor systems.

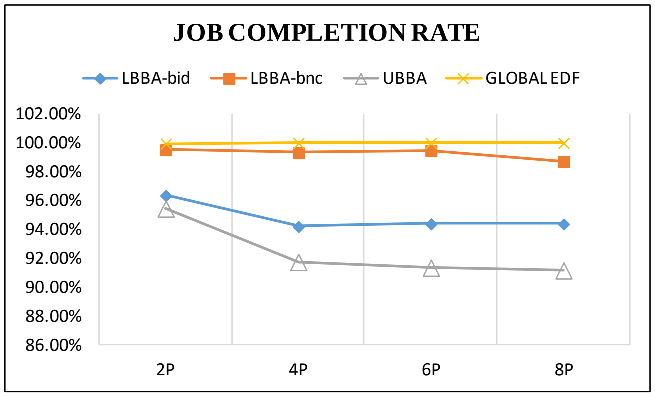

Fig. 5. Average job completion percentage on average by LBBA-bid, LBBA-bnc, UBBA and Global EDF, for 2, 4, 6, and 8 processors

Figure 5, demonstrates the schedulability of the algorithms in comparison with Global EDF with almost 100% job completion rate. The algorithms with implicit deadlines (LBBA-bid and UBBA) had the same rate of 96% on 2 processors which slightly decreased to 94% on 8 processors in LBBA-bid. Considering that these results are achieved for worst-case execution times of the tasks, in actual soft real-time applications this minor amount of tardiness would be sustainable. The rate for UBBA decreased to 92% for 4, 6 and 8 processors.

On the other hand, LBBA-bnc with non-implicit deadline showed substantial improvement in job completion rate and appeared very close to optimal schedulability (99.5% to 99.8%).

V. Conclusion And Future Work

In this paper, we have proposed a new semi-partitioning approach to schedule soft real-time periodic task sets on identical multiprocessor systems. This method allows task migration at job-boundaries, i.e., different instances (or jobs) of each task can be assigned to any of the processors in the system at their release time. However, after they are partitioned, no migration is allowed. We used our hybrid scheduling method, LBBA, which maximizes total benefit while balancing the workload among the processors for lowering cost and reducing tardiness. We have demonstrated these advantages of LBBA (for aperiodic tasks) in our previously published papers. In this paper, we provided the upper bounds on the delays and preemptions in accordance to the task utilizations, and schedulability conditions of periodic tasks.

In this work, we studied the performance of our model in RTS systems with implicit and non-implicit deadlines, in terms of total gained benefit, job completion rate and total number of preemptions.. In addition, we proposed a modified version of our model to balance the task utilizations among the processors instead of their execution times. We have conducted experimental performance analysis of the LBBA algorithm for periodic tasks with implicit deadlines, along with two propositions (LBBA-bnc with non-implicit deadlines, and UBBA with utilization-balancing) for more benefit accrual and higher percentage of completed jobs.

In order to evaluate their performance, we considered metrics such as total gained benefit, schedulability in the term of job completion rate, and total number of preemptions. The experimental results show that LBBA for the tasks with non-implicit deadlines is near optimal, with the same performance on benefit maximization as in the other two methods with implicit deadlines. Also, our algorithms have fewer numbers of preemptions than Global EDF as the number of processors increases.

For the future work, these algorithms can be compared to the state-of-the-art in semi-partitioning such as EDF-os and EDF-fm, and also to other benefit-based scheduling algorithms.

References

[1] A. Elnably, K. Du, P. Varman, "Reward scheduling for QoS in cloud applications," in *Proceedings 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* vol. 29, pp. 1645–1660, 2013.

[3] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *Proceedings 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," *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," *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," *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 *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, 2011.

[8] B. Awerbuch, Y. Azar, and O. Regev, "Maximizing job benefits online," in *Proceedings 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 *Proceedings 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 *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* August 20-22, 2014.

[12] B. Sanati, A.M.K. Cheng, "LBBA: An efficient online benefit-aware multiprocessor scheduling for QoS via online choice of approximation algorithms," *Future Generation Computer Systems*, vol. 59, pp. 125–135, June 2016, (Available online December 2015).