An Efficient Online Benefit-aware Multiprocessor Scheduling Technique for Soft Real-Time Tasks Using Online Choice of Approximation Algorithms

----

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Behnaz Sanati

December 2016

An Efficient Online Benefit-aware Multiprocessor Scheduling Technique for Soft Real-Time Tasks Using Online Choice of Approximation Algorithms

Behnaz Snati

APPROVED:

Acknowledgements

During my PhD studies, I have met many people whose presence was essential for my achievements and success; therefore, I dedicate the following humble words to express my gratitude and to acknowledge their help each in their own way.

I would like to start by thanking my advisor, Dr. Albert Cheng, for his advice and constant encouragements during the entire period of my research work. I would like to thank my Doctoral committee members, Dr. Lennart Johnsson, Dr. Ioannis Pavlidis, Dr. Motgomery Pettitt, Dr. Ernst Leiss and Dr. Jaspal Subhlok for their helpful comments and suggestions. Among these professors, I would especially like to thank Dr. Ernst Leiss for always being available to answer my questions and also for reviewing and proof-reading my dissertation.

Many thanks to the members of our RTS Lab, Youngme Lee and Nicholas Troutman for their assistance in implementing the algorithms, and also Xingliang Zou and Carlos Rincon for reviewing our research paper and sharing helpful comments. I would like to thank Yvette Elder, Liz Faig and the administration staff for their constant help.

I am fortunate to have a loving family who supports me all the time and has faith in me. I am thankful for my parents, who are by far the strongest persons I have ever seen and who taught me how to face challenges with patience and persistence, and my brothers for being always there for me. I learned a lot from this experience and I believe that I could not have succeeded in my work without their support and encouragement. I am very grateful to my husband and our precious daughter, Mona. I will never forget their supportive and understanding attitude when I am overloaded with work. Their presence in my life is a great blessing.

Finally, I am thankful for Allah the almighty, my success can only come from Him.

To my family, and to my husband and daughter

for their

love, endless support and encouragement …

An Efficient Online Benefit-aware Multiprocessor Scheduling Technique for Soft Real-Time Tasks Using Online Choice of Approximation Algorithms

---

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Behnaz Sanati

December 2016

Abstract

Maximizing the benefit gained by soft real-time tasks in many applications and embedded systems is highly needed to provide an acceptable QoS (Quality of Service). Examples of such applications and embedded systems include real-time medical monitoring systems, video-streaming servers, multiplayer video games, and mobile multimedia devices. In these systems, tasks are not equally critical (or beneficial). Each task comes with its own benefit-density function which can be different from the others'. The sooner a task completes, the more benefit it gains. In this work, a novel online benefit-aware preemptive approach is presented in order to enhance scheduling of soft real-time aperiodic and periodic tasks in multiprocessor systems. The objective of this work is enhancing the QoS by increasing the total benefit, while reducing flow times and deadline misses. This method prioritizes the tasks using their benefit-density functions, which imply their importance to the system, and schedules them in a real-time basis. The first model I propose is for scheduling soft real-time aperiodic tasks. An online choice of two approximation algorithms, greedy and load-balancing, is used in order to distribute the low-priority tasks among identical processors at the time of their arrival without using any statistics. The results of theoretical analysis and simulation experiments show that this method is able to maximize the gained benefit and decrease the computational complexity (compared to existing algorithms) while minimizing makespan with fewer missed deadlines and more balanced usage of processors. I also propose two more versions of this algorithm for scheduling SRT periodic tasks, with implicit and non-implicit deadlines, in addition to another version with a modified load-balancing factor. The extensive simulation experiments and empirical comparison of these algorithms with the state of the art, using different utilization levels and various benefit density functions show that these new techniques outperform the existing ones. A general framework for benefit-aware multiprocessor scheduling in applications with periodic, aperiodic or mixed real-time tasks is also provided in this work.

Table of Contents

List of Figures

List of Tables

1. Online Benefit-aware Multiprocessor Scheduling of Aperiodic Real-Time Tasks

1.1. Introduction and Motivation

Multiprocessor platforms are widely adopted for many different applications in embedded systems and server farms. They are becoming even more popular since many chip makers including Intel and AMD are releasing multi-core chips. Adopting multiprocessor platforms can enhance the system performance, but scheduling jobs optimally on a multiprocessor system is an NP-hard problem [1], [2].

There are two major models for this scheduling problem. The first is the cost model; its goal is to minimize the *cost*, which is the *overall flow time*, also referred to as *response time* or *makespan*. The second model is the benefit model which aims to maximize the benefit of jobs that meet their deadlines. The latter model is used for soft real-time applications in which deadline misses are sometimes tolerable. Examples of such applications using multi-core platforms are multi-purpose home appliances such as HDTV streaming and interactive video games. More motivating examples from the domains of multimedia, air defense and enterprise-level, asynchronous, cooperating real-time computer systems are given by Welch and Brandt [3]. Other examples of such applications and embedded systems multimedia applications as explained in [4], image and speech processing [5], [6], [7], [8], time-dependent planning [9], robot control/navigation systems [10], [11], medical decision making [12], information gathering [13], real-time heuristic search [14], database query processing [15], and the Internet of Things (IoT) [16].

In this research, the main focus is on the benefit model to maximize the benefit of online preemptive scheduling of soft real-time jobs on multiprocessor systems having identical processors. I propose an online choice of two approximation algorithms, greedy and Load-Balancing, to reduce the cost or makespan. More balanced distribution of the jobs between processors in this novel approach results in a lower overall flow time, less total idle time, fewer missed deadlines, and more efficient usage of CPU cycles. Also, this model eliminates the runtime overhead of migrating jobs by prohibiting migration and adapting a partitioning approach, instead.

In the following subsections, I provide an overview of previous works on approximation algorithms and maximizing benefit on-line for multiprocessors. Section 1.2 presents the details of the proposed approach and an example to illustrate its differences from the previous methods. The analysis of the new method is provided in Section 1.3. Section 1.4 contains the performance metrics and the experimental settings and results. Finally our conclusions are presented in Section 1.5.

1.1.1. Related Work on Approximation Algorithms

Approximation algorithms are often used to attack difficult optimization problems, such as job scheduling on multiprocessor systems which is an NP-hard problem [1], [17], [18]. An approximation algorithm settles for non-optimal solutions found in polynomial time, when it is very unlikely to find an efficient exact algorithm to solve NP-hard problems, or the sizes of the data sets are so large that they make the polynomial exact algorithms too expensive.

A greedy 2-approximation algorithm is used in [19] for fault tolerance and in [20] for benefit maximization in identical multiprocessor systems. Even though greedy approximation can be a good solution in many cases, a load balancing approximation can result in a shorter flow time for a set of jobs when we have to distribute several jobs among multiple processors at the same time [21], [22]. Piel et al. [23] have proposed a load balancing technique based on statistics for real-time scheduling on asymmetric multiprocessors. They apply partitioning for high priority jobs and migration for jobs with low priority.

1.1.2. Related Work on Benefit-aware Real-Time Computing

The gained benefit can vary when using different benefit functions. Researchers have investigated applying benefit functions for allocating resources in limited, soft real-time systems [24], [25], [26]. Andrews et al. [27] provided a framework to formalize the use of benefit functions in complex real time systems.

Buttazzo et al. [28] provided the results of studying jobs that are characterized by an importance value. The performance of the scheduling algorithm was then evaluated by computing the cumulative value (or benefit) gained on a job set. However, the target of their research was uniprocessor scheduling. Welch et al. [3] discussed how benefit is used in a variety of real-time paradigms and in example applications. Awerbuch et al. [29] presented a constant competitive ratio algorithm for a benefit model of on-line preemptive scheduling. This method can be used on both uniprocessor and multiprocessor systems. Aydin et al. [30] proposed a reward-based scheduling method for periodic real-time tasks and [31] presented online scheduling policies for a class of IRIS (Increasing Reward with Increasing Service) real-time tasks.

1.2. Online Choice of Approximation Algorithms

The algorithm proposed in [29] only focuses on maximizing the total benefit gained without being concerned with minimizing the overall flow time of a job set (*response time* or *makespan*). In that method, the benefit gained by each job that completes its execution is calculated using the benefit density function of its flow time. This function is a non-increasing, non-negative function of time, by definition [29]. It means the more the flow time, the less the benefit gained.

Therefore, we proposed, simulated and analyzed an efficient online benefit-aware technique with choices of approximation algorithms including greedy and Load-balancing to distribute jobs among multiple processors at the time of release [21]. This method prioritizes the jobs using their benefit density functions and schedules them on a real-time basis in order to reduce the makespan (overall flow time) of the jobs and total idle time of the processors while maximizing the total gained benefit.

I also used the online choice of two approximation algorithms (greedy and load-balancing) as a solution for special cases that were not considered in the existing benefit-aware multiprocessor scheduling algorithms such as the *Benefit-Based Algorithm* proposed in [29] which we refer to as BBA in the rest of this dissertation. Examples of those cases are when there are several high priority jobs which can preempt a running job or when a high priority job can preempt more than one running job.

In order to be able to balance the workload among the processors by partitioning the jobs as soon as they are released, a separate pool of the waiting jobs is considered for each processor. This method is referred to as *Load-Balancing/Greedy Benefit-Aware* algorithm (or LBBA) throughout the dissertation. This load-balancing technique is different from what Piel et al. [23] have used, since we do not use statistics for distributing the jobs. Instead, decisions are made online by using the actual (worst case) execution times of ready jobs and the remaining workload of the processors for partitioning on a real-time system with identical processors. Migration is not allowed in our proposed real-time system model.

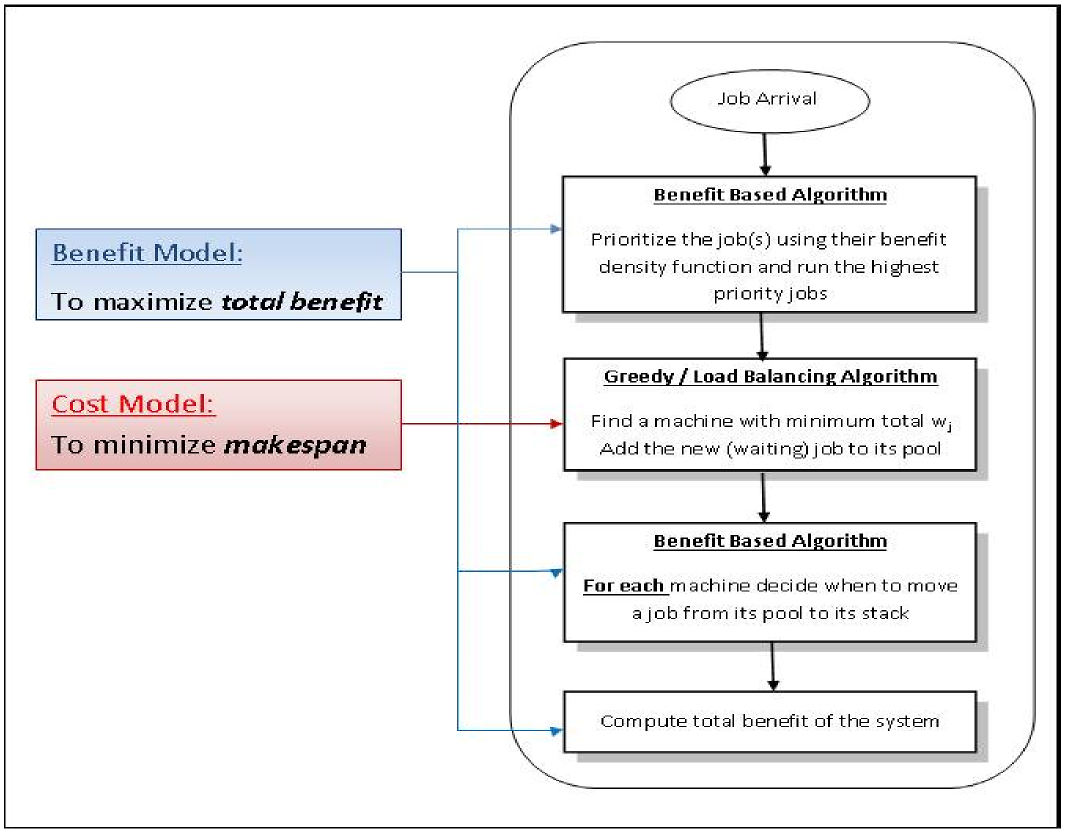

Job Arrival
Benefit Based Algorithm
Prioritize the job(s) using their benefit density function and run the highest priority jobs
Benefit Model:
To maximize total benefit
Cost Model:
To minimize makespan
Greedy / Load Balancing Algorithm
Find a machine with minimum total w_j
Add the new (waiting) job to its pool
Benefit Based Algorithm
For each machine decide when to move a job from its pool to its stack
Compute total benefit of the system

Fig. 1.1: LBBA Methodology

LBBA is superior to BBA in principle, since:

- **LBBA is a novel hybrid model** of soft real-time multiprocessor scheduling. In contrast to BBA, which only follows a benefit model, LBBA is a combination of benefit and cost models. That is, it aims to **minimize makespan** in order to achieve the *maximum benefit at the lowest cost*.
- **No synchronization is needed** for fetching the jobs from a shared pool. That is because a separate pool is assigned to each processor in contrast to the other method (BBA) where all processors use a shared pool.
- **LBBA facilitates load-balanced partitioning** of waiting jobs, while this case is not considered in BBA. For example, in case the waiting (or ready) jobs arrive *asynchronously*, LBBA adapts the "greedy approximation" to assign a job to the pool of the processor with the least remaining workload. If jobs are *synchronous*, i.e., arrive at the same time, those that cannot start running and have to wait in a pool, will be partitioned among the processors, using our "Load-Balancing" technique.
- **LBBA optimizes the CPU usage and minimizes the total idle time** of the processors by balancing the workload among them.
- **LBBA improves Quality-of-Service (QoS) by reducing missed deadline ratio:** As shown by an example in 1.2.4, LBBA reduces the possibility of starvation for low priority jobs, comparing to BBA. It also has a **Minimal Response time,** including both scheduling and execution time, for a job set (up to 300 times faster response time than BBA in our experiments shown in 1.4.1.2.).

- **LBBA is computationally less expensive** than BBA, as we prove in sub-section 3.2.

The reasons for which we are applying both greedy and load-balancing approximation algorithms, depending on the situation, instead of only one of them all the time, are as follows:

a) *Why we do not use greedy approximation all the time:*

The *Greedy-Benefit-Based* model, called GBBA, that we proposed in [20] in the early stage of this research handles the special cases we mentioned earlier, that were overlooked by BBA, by applying a 2-approximation greedy method. However, it has a shortcoming in minimizing makespan when several ready jobs are going to be partitioned among the pools of the processors at the same time. Also, this is the case when several synchronous high priority jobs can preempt more than one running job. To overcome this problem, we add our Load-Balancing approximation method to GBBA. Our hypothesis is that the online combination of these two approximation methods can minimize the makespan while maximizing the total benefit.

b) *Why we do not use Load-Balancing all the time*:

In Load-Balancing, we sort the jobs in descending order of their workload and the processors in ascending order of their total remaining workload (both on the stack and the pool of each processor). Then the first job in the TempList (list of the ready jobs), which has the heaviest workload, will be assigned to the first processor in the list, having the least remaining workload. The second job in the list will be assigned to the second processor, and so on.

This method, by itself and without the help of greedy approximation, can be used to partition the jobs among the processors so that the distributed workload is as balanced as possible. Greedy approximation does not facilitate Load-Balancing, when many-to-many assignments are needed. However, in order to optimize computation time, we use the greedy method in cases where a one-to-many or many-to-one assignment is needed.

In this research, both BBA and our solution, LBBA, are simulated extensively in order to compare their performances. Figure 1.1 demonstrates the LBBA hybrid methodology which is a combination of benefit model and cost model. In the following sub-sections, we define our system and task model, and the notations used in our algorithm, along with the detailed explanation of LBBA algorithm. At the end of this section, we illustrate the advantages of LBBA over BBA and GBBA through an example.

1.2.1. System and Job Model

A multiprocessor system with *m* identical processors is considered for our partitioning approach. In the partitioning approach no migration of jobs is allowed. Therefore, each job has to stay with only one processor during its whole execution time. This method is possible if each processor has its own pool instead of sharing a pool with other processors.

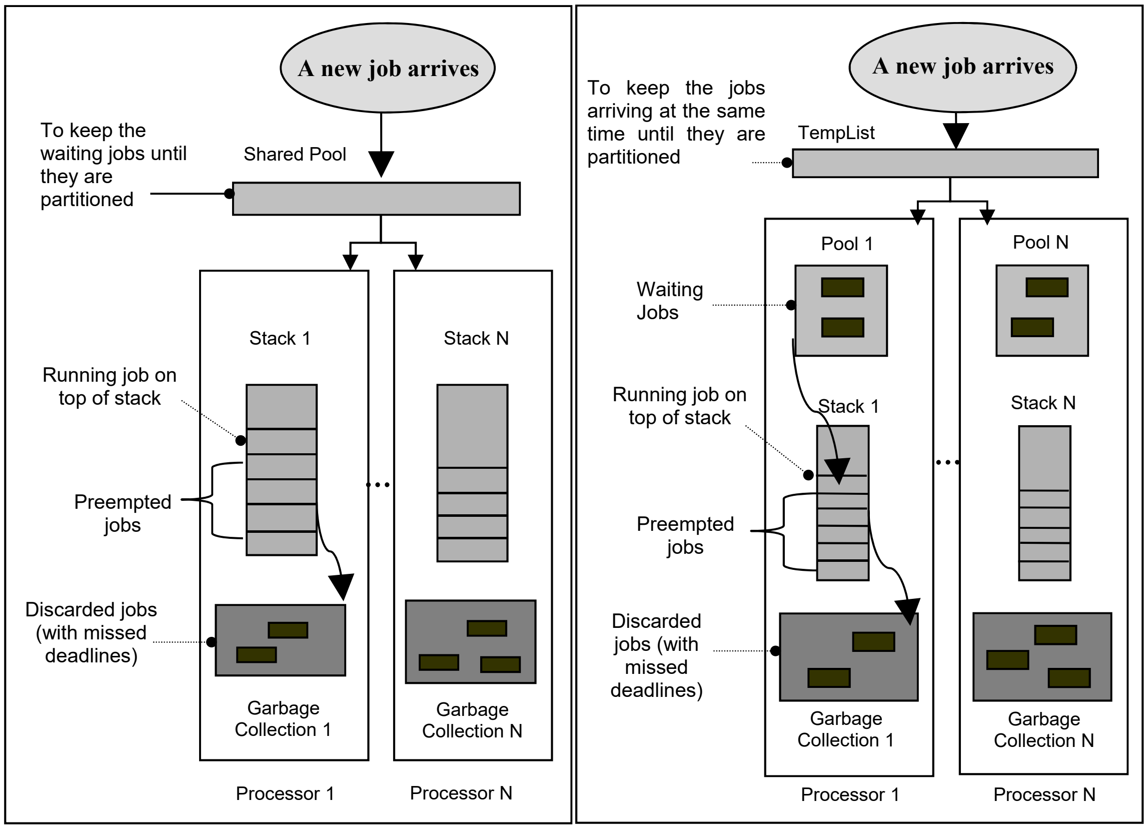


Fig. 1.2: Job storage locations with a shared pool in BBA (left) and a separate pool for each processor in LBBA (right)

Also, as Figure 1.2 shows, each processor has its own stack and garbage collection. This chapter explores the scheduling of aperiodic soft real-time job sets which are independent in execution and there are no precedence constraints among them. Pre-emption is allowed. Each aperiodic job may be released at any time. An example of such aperiodic jobs is a partial air defense subsystem as mentioned by Welch et al. [3].

1.2.2. Notation

The definitions of our notation are as follows:

r_j – *release time* of job J_j

w_j – *worst case execution time (WCET)* of job J_j, simply considered as execution time or *workload* of job J_j in this paper

s_j – *start time* of job J_j

c_j – *completion time* of job J_j

Br_j – *break point* or *deadline* of job J_j,

$$Br_j = s_j + 2w_j \tag{1}$$

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

$$\frac{\beta_j}{\beta_j(t+w_j)} \leq C \tag{2}$$

Note: for $t < w_j$, there would be no benefit gained by job J_j, since it has certainly not completed its execution at time *t*.

f_j – *flow time* of job J_j:

$$f_j = c_j - r_j \tag{3}$$

b_j – *benefit,* gained by a completed job J_j:

$$b_j = w_j.\ \beta_j(f_j) \tag{4}$$

$d_j(t)$ – *variable priority* of job J_j at time *t,* before scheduling ($t < s_j$):

$$d_j(t) = \beta_j(t + w_j - r_j) \tag{5}$$

d'_j – *fixed priority* of job J_j, when it is scheduled and starts its execution:

$$d'_j = \beta_j(s_j + w_j - r_j) \tag{6}$$

1.2.3. LBBA Algorithm

LBBA is adopting the same definition of breakpoint, benefit density function, priority of the jobs in pools or on the stacks as used in BBA, and also the same preemption condition to be able to determine if it could improve that algorithm by applying approximation algorithms and necessary modifications to the system. However, to show that this solution does not sacrifice benefit maximization in order to obtain the minimum response time, we prove that the competitiveness of BBA is also preserved in LBBA algorithm.

A desired property of the system in this method is the possibility to delay jobs without drastically reducing overall system performance. Also, this algorithm does not use migration on the multiprocessor system. The LBBA algorithm is an event-driven algorithm. 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 partitioned among the processors.

Each job J_j arrives with its own execution time (w_j) and benefit density function $B_j(t)$ for ($t \geq w_j$). The flow time of a job, denoted by f_j , is the time that passes from its release time (r_j) to its completion time (c_j); it is at least equal to w_j (execution time). The benefit gained by each job that completes its execution is a function of its flow time (Equation 4). The job on top of each stack is the job that is running and all other jobs in the stacks are preempted. If a job reaches its breakpoint 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. This means the break point of a job is its deadline, which is twice its execution time after it starts running.

The priority of each unscheduled job (located in each pool) at time t which is denoted by $d_j(t)$ (for $t \leq s_j$) is variable with time. However, for $t > s_k$ (when the job k has started its execution) the priority is calculated as $d'_k = B_k (s_k + w_k - r_k)$ (lines 19 and 68 of the following pseudo-code, Algorithm 1). The notation d'_k is used for the priority of the running job J_k on top of the stack. This priority is given to the job J_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 fixed values, the priority of a job will not change after its start time.

Once a new job J_j is released, if there is a processor such that $d_j(t) > 4d'_k$ (lines 58 through 66), or its stack is empty (lines 11 through 22), then the newly released job is pushed onto the stack and starts running, otherwise it will be partitioned among the pools of the processors using an online choice of load balancing or greedy approximation (lines 39 through 75). Awerbuch et al. [29] used the preemption condition ($d_j(t) > 4d'_k$) and their analysis shows that the factor 4 in this condition plays the role in the BBA constant ratio competitiveness being equal to $10C^2$. Therefore, in order to preserve this competitiveness, we use the same criterion. Later in the analysis of our algorithm, we prove how this competitiveness is preserved by LBBA.

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 if $d_j(t) \leq 4d'_k$ for all J_j in its pool, otherwise, it gets the job with max $d_j(t)$ 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 garbage collection. If a job completes before reaching its break point, its gained benefit is calculated and added to the total benefit. 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).

To be able to assess the performance of LBBA, we need to consider various situations of the released jobs, regarding their release times and workloads. Here, I discuss different scenarios and how they are handled. In addition, I provide an example in which BBA can result in a very long waiting time for some jobs before they get scheduled or even their starvation. Analysis shows that LBBA overcomes this problem and that is one of the key aspects of LBBA which reduces the missed deadline ratio and improves the Quality of Service.

Case 1: J_j is a newly released job at time t

Lemma 1: *For a newly released job, J_j, at time t, its priority is independent of its release time, r_j, but relies on its workload, w_j.*

Proof: Since J_j is released at time t,

$$t = r_j \tag{7}$$

From the equations (5) and (7):

$$d_j(r_j) = B_j(w_j) \tag{8}$$

So, Lemma 1 is proved. Equation (8) shows that the priority of job J_j at its release time is a function of w_j, regardless of its release time, r_j. ■

Case 2: J_i is a waiting job in a pool

Lemma 2: *If J_j cannot start its execution at its release time, it has to wait in pool, then its variable priority $d_j(t)$ will not increase at any time t, ($r_j < t < s_j$) while it is waiting.*

Proof: At any time instance *t, while* J_j is waiting $t > r_j$. By definition, $d_j(t) = B_j(t + w_j - r_j)$ and also B_j is a non-increasing, non-negative function. Hence, Lemma 2 is proved:

$$\text{For all } t,\ t > r_j, \qquad d_j(t) \le d_j(r_j) \tag{9}$$

■

Theorem 1: *If J_i is released and cannot be scheduled at release time by BBA, if the next jobs have the same workload as J_i or less, J_i may starve or wait until all of them are scheduled.*

Proof: Based on Lemma 1 and Lemma 2, if J_i is waiting in the shared pool (in BBA method) when J_j is released, then at $t \ge r_j$, its priority will be less than the priority of J_j. So, if any processor is available or the priority of J_j is high enough to preempt another job, then J_j is scheduled before J_i. If the next released jobs all have the same or a smaller workload than J_i does, then it has to wait in the shared pool until all of them are scheduled. ■

ALGORITHM 1: LBBA (for aperiodic tasks)

```
1  Required: One or more jobs arrive at time $t \geq 0$
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 j in the
8          TempList:
9          $d_j(t) = B_j(t + w_j - r_j)$
10     Sort TempList based on the priority
11     If (at least one stack is empty)
12     {
13       Push the highest priority job(s) j onto
14           empty stack(s) of idle processor(s) i;
15       Add its execution time $w_j$ to total workload
16           of the stack of the processor i ($\sum W_{si}$),
17       Recalculate total workload of processor i:
18           $W_i = \sum Wp_i + \sum Ws_i$
19       Calculate the fixed priority of j using its
20           start time $s_j$:
21           $d'_j(t) = B_j(s_j + w_j - r_j)$
22       Start executing j,
23     }
24     Else
25     {
26        /* no stack is empty */
27        /* preempt if possible otherwise
28           distribute among the pools */
29       Compare the priority of the ready jobs in
30           TempList with the priority of the running
31           running jobs (indicated by index k)
32           onto the stacks:
33       If ($d_j(t) \leq 4d'_k$ for ( each job j in TempList
34           and each running job 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 j in TempList)
40         {
41             Sort the processors in ascending order
42               of their total remaining workload
43               on their pools and stacks :
44               $W_i = \sum Wp_i + \sum Ws_i$
45             Append the job j with largest
46                 execution time $w_j$ to the pool of the
47                 processor i with minimum remaining
48                 workload;   /* load balancing */
49             Remove j from TempList;
50             Add its execution time $w_j$ to total
51                 workload of the pool of processor i
52                 ($\sum Wp_i$);
53             Recalculate total workload of
54                 processor i:
55                 $W_i = \sum Wp_i + \sum Ws_i$
56         }
57       }
58       Else
59           /* if ($d_j(t) > 4d'_k$)  then ( j preempts k)*/

   Greedy Approximation (multiple-choice Preemption)

60           /* If 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 j onto the stack on top of k,
67           Start executing j,
68           Calculate the fixed priority of j using its
69               Start time $s_j$,: $d'_j(t) = B_j(s_j + w_j - r_j)$
70           Add the execution time of j to the total
71               workload of that stack ($\sum Ws_i$ ),
72           Recalculate total workload of the
73               Processor i:
74               $W_i = \sum Wp_i + \sum Ws_i$
75       }

   Check for missed Deadlines

76       /* at each time instance t, if any of the
77          running jobs on top of the stacks has
78          reached its break point ($t > Br_j$),
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 j calculate $b_j$ */
84       $b_j = w_j.\ \beta_j(f_j)$
85     }

   Total Benefit Calculation

86       /* calculate the sum of all benefits gained,
87          q being the number of completed jobs */
89       $B = \sum_{j=1}^{q} bj$
90  }
```

In LBBA, J_i instead of being kept in the shared pool, will be assigned to the pool of a processor based on greedy or load-balancing method, depending on the situation. Also, the next released jobs will not all get assigned to the same pool and will be distributed among all processors. This means the waiting time of J_i will be significantly less in LBBA than BBA method.

Theorem 2: *If J_j cannot preempt any currently running jobs at its arrival (i.e, release time), then it will not be able to preempt any jobs that start running after release of J_j while J_j is waiting.*

Proof: There will be two different scenarios for this situation. Hence, Theorem 2 can be deduced from the two following Lemmas. ■

Lemma 3: *Let J_j be a job that is waiting in a pool. If J_s is released after J_j ($r_s > r_j$), and is scheduled before J_j, then it cannot be preempted by J_j at any time during its execution.*

Proof: J_s is released at r_s, after release of J_j (i.e., $r_s > r_j$); J_s is scheduled and starts its execution, while J_j is waiting at s_s (start time of J_s); therefore,

at $t_1 = s_s$, $$d'_s > d_j(t_1) \tag{10}$$

$$d'_s > d_j(s_s) \tag{11}$$

Based on the priority assignment rule of the algorithm, d'_s, the priority of J_s at time s_s is a fixed priority and will not change with time, for $t \geq s_s$. However, the priority of J_j which is still waiting, will not increase:

For $t_2 \geq t_1$, $$d_j(t_2) \leq d_j(t_1) \tag{12}$$

$$d_j(t_2) < d_j(s_s) \tag{13}$$

From 8 and 10: $d_j(t_2) < d'_s$ (14)

Hence, J_j will not be able to preempt J_S at any time after J_S starts running, due to its priority not being high enough to preempt J_S. ■

Lemma 4: *If J_j is a waiting job, it cannot preempt any running job J_p($r_p \leq r_j$) which was scheduled either before J_j was released ($r_p < s_p < r_j$) or when J_j was released ($r_p \leq r_j \leq s_p$).*

Proof: From Lemma 1, if J_j cannot preempt J_p at $t = r_j$, assuming J_p was running at r_j, it will not preempt J_p at any time $t > r_j$, since $d'_p > d_j$ for all time instances $t \geq r_j$.

Also, if J_p was released at the same time or before J_j was released at $t = r_j$, and J_p is scheduled before J_j ($r_p \leq r_j \leq s_p$), it shows that:

At $t = s_p$, $d_p(t) > d_j(t)$ (15)

$d'_p = d_p(s_p)$ (16)

Therefore, $d'_p > d_j(s_p)$ (17)

Hence, for $t > s_p$:

From (12): $d_j(t) \leq d_j(s_p)$ (18)

From (17) and (18): $d_j(t) \leq d'_p$ (19)

So, J_j will not be able to preempt J_S in this case and Lemma 4 is proved. ■

1.2.4. An Example

The following example, provided in Table 1.1, is a set of independent, real-time jobs which contains both synchronous and asynchronous jobs. The WCET (Worst-Case Execution Time) of each job will be known when it arrives. Figure 1.3 shows how this job set will be scheduled by BBA, GBBA and LBBA on a 3-processor system. The benefit gained by each completed job is calculated using $\beta_j(t) = 1/(2w_j)$ as the benefit density function, and shown in Table 1.1:

Table 1.1: An example of a job set and job benefits gained by BBA, GBBA and LBBA

Job ID	Arrival Time (r_j)	Execution Time (w_j)	Benefit Gained by		
			BBA	GBBA	LBBA
a	0	8	0.00	0.00	0.33
b	1	4	0.40	0.40	0.40
c	1	6	0.50	0.43	0.43
d	2	3	0.21	0.25	0.25
e	3	2	0.50	0.50	0.50
f	3	4	0.25	0.22	0.15
g	3	6	0.21	0.27	0.27
h	6	2	0.50	0.50	0.50
i	6	1	0.50	0.50	0.50
j	8	2	0.50	0.50	0.50
k	9	3	0.50	0.25	0.25
l	10	2	0.50	0.50	0.50

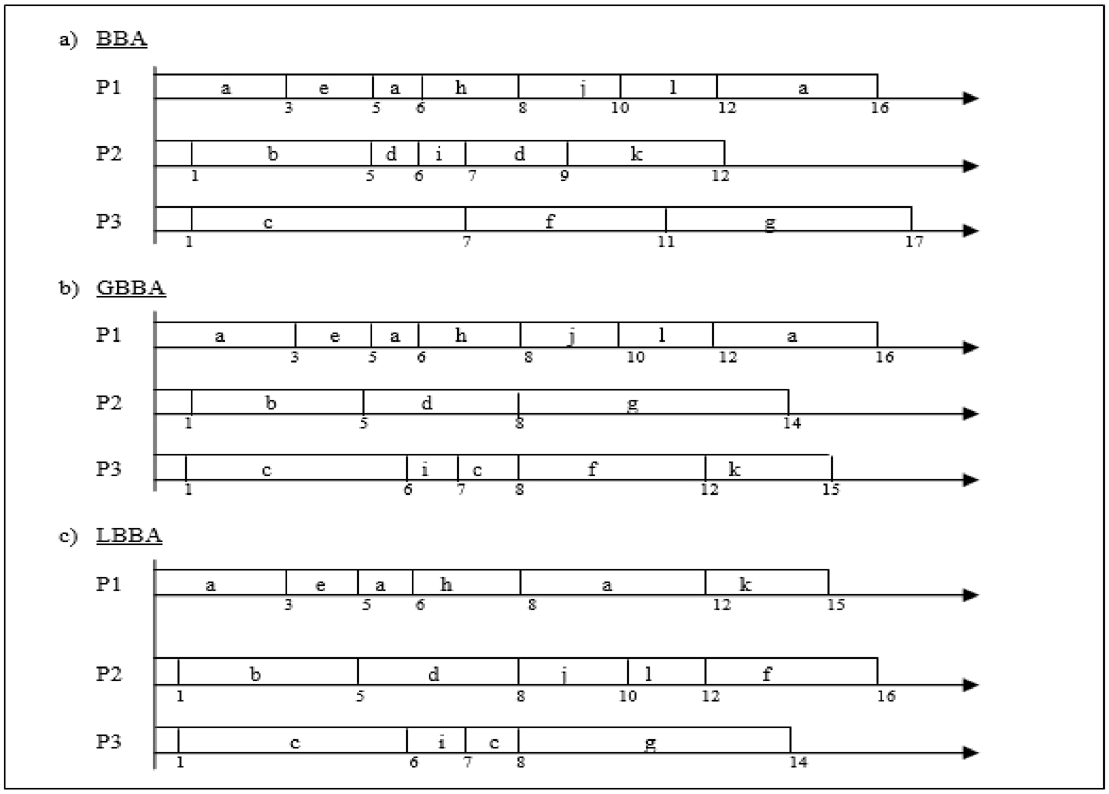


Fig. 1.3: Job scheduling on a 3 processor system by BBA, GBBA and LBBA

Table 1.2: Comparing the performance of BBA, GBBA and LBBA

Algorithm	Missed Jobs	Preemptions	MakeSpan	Total Idle Time	Total Benefit	Benefit-to-Cost Ratio
BBA	1	5	17	8	4.57	0.269
GBBA	1	5	16	5	4.32	0.270
LBBA	**0**	**3**	**16**	**5**	**4.58**	**0.286**

We can summarize the performance of the three scheduling methods considering the following metrics:

- **Preemptions:** When job *a* is scheduled by BBA and GBBA schedulers, it gets preempted twice by higher priority jobs *e* and *h,* also kept in preemption by jobs *j* and *l.* It completes its execution right at its break point which is at time *t* = 16. Therefore, job *a* does not gain any benefit, even though the system has fully executed it, and it is considered as a missed job in both BBA and GBBA. However, it is only preempted by jobs *e* and *h* under LBBA scheduling. Job *i* preempts job *d,* in BBA, while it preempts job *c* in both GBBA and LBBA.

- **Makespan:** LBBA and GBBA have shorter makespan than BBA.
- **Total Benefit:** LBBA not only preserves the benefit maximization aspect of BBA, but exceeds it. Also, it improves GBBA in this regard.
- **Total Idle Time:** In LBBA, total idle time was 38 % less than BBA and GBBA.
- **Benefit-to-Cost Ratio:** As shown in Table 1.2, the benefit-to-cost ratio of LBBA is higher than BBA and GBBA and it is improved about 6.3 % in this example.
- **Missed Ratio:** The ratio of missed jobs in BBA is 8.33 % while in GBBA and LBBA was 0.00 %.

This example demonstrates how LBBA can improve the QoS, comparing two other state of the art benefit-aware methods, BBA and GBBA, by:

- Maximizing total benefit
- Maximizing benefit-to-cost ratio
- Minimizing total idle time
- Minimizing makespan
- Minimizing missed ratio

1.3. Analysis

In order to evaluate our proposed algorithm, we analyze it from two points of view: computational complexity and benefit maximization.

1.3.1. Job Benefit Maximization

BBA is proved in [29] to be a constant competitive ratio algorithm ($10C^2$) for both uniprocessor and multiprocessor scheduling. This is with considering the restriction shown in equation (2), $\beta_j / \beta_j(t + w_j) \leq C$, to be satisfied for each $\beta_j(t)$ and some fixed constant C. That is, in case of delaying a job by its length, we only lose a constant factor in its benefit.

In order to preserve the competitiveness of that algorithm, we are adopting the same definition of breakpoint, benefit density function and its restriction, priority setting for the jobs in the pools or on the stacks, and also the same preemption condition. Therefore, the same proof of that competitiveness ($10C^2$) is true for each processor the same way as for BBA uniprocessor scheduling. That is because after partitioning, no migration is allowed and we have uniprocessor scheduling for the set of jobs on the pool of each processor based on the priority of the jobs; also the running job on each processor can be preempted by a newly arrived high priority job (in TempList) if the preemption condition is satisfied.

If m denotes the number of processors, i, the index of a processor, v_i^{OPT}, total benefit of each processor by optimal scheduler and v_i^{LBBA}, total benefit of each processor by LBBA:

$$v_i^{LBBA} \geq \frac{1}{10C^2} v_i^{OPT} \tag{20}$$

Then, adding up the benefits gained by all the processors in the system will result in:

$$\sum_{i=1}^{m} v_i^{LBBA} \geq \frac{1}{10C^2} \sum_{i=1}^{m} v_i^{OPT} \tag{21}$$

Now, let V^{OPT} denote the total benefit gained by the optimum scheduling of a set of jobs and V^{LBBA} the total benefit gained by LBBA for the same job set. Then equation (22) shows that algorithm LBBA is also $10C^2$ competitive:

$$V^{LBBA} \geq \frac{1}{10C^2} V^{OPT} \tag{22}$$

1.3.2. Computational Complexity

In the BBA method, at each time step, the priority of all jobs in the shared pool must be compared with the priority of the running jobs on the top of all processor stacks. If there are m processors in the system and n waiting jobs in the pool, then n times m comparisons are needed at each time step to determine if any of the waiting jobs can be pushed onto any stack and start running.

On the other hand, our method performs (m - 1) comparisons at each job arrival to find the least utilized processor and adds the execution time of new job j to its utilization for future comparisons, resulting in m operations at each job arrival. However, if r jobs arrive at the same time ($r > 1$), a load-balanced/greedy partitioning is performed: The jobs will be sorted based on a non-increasing order of their execution times, which roughly needs $r \log_2 r$ comparisons. Then, the first job in the list is assigned to the pool of the processor with the least remaining workload, and so on.

At each time step, if x_1 is the number of waiting jobs in first pool, x_2 in the second pool, and so on, then X denotes the total number of waiting jobs ($X = x_1 + x_2 + \ldots + x_m$). Since our new method only compares the priorities of waiting jobs in each pool with the priority of the running job on the corresponding stack, only X comparisons are done at each time step. It is now clear that our method is computationally less expensive than the original one.

In the next section, the results of our extensive experiments are provided which show a significant improvement in the scheduling speed by our method especially for systems with very large work load.

1.4. Performance Evaluation

Schedulability is one of the main performance metrics to evaluate a scheduler for hard real-time systems. However, benefit-aware schedulers are mainly used for soft real-time systems, in which missing a deadline would not drastically affect the performance. For soft real-time scheduling, the total value or benefit gained and also the miss ratio are the performance metrics. In this research [32], we considered the following measurements to evaluate and compare the performance of both BBA and LBBA algorithms:

- The benefit gained by completed jobs
- Missed deadline ratio
- The benefit-to-cost ratio
- Total processor idle time

The cost is the overall flow time of a job set. It is the time that has passed since the first job has arrived till the last job is completed. The benefit-to-cost ratio is calculated by dividing the total benefit by the overall flow time.

Reducing flow time and using processors more efficiently are other goals addressed by our solution. Therefore, to measure how efficiently CPU cycles have been used by either algorithm, we have considered total processor idle time. Total processor idle time is obtained by accumulating all the time periods in which any of the processors has been idle. The more the total idle time decreases while preserving or even improving the amount of gained benefit, the more efficiently processors are used and the shorter the response time or overall flow time (or cost) gets.

1.4.1. Experimental Settings and Results

In order to evaluate the performance of our method (LBBA) and compare it with the performance of the previous one (BBA), we implemented both algorithms in C++ to simulate the scheduling of synthetic job sets using different numbers of identical processors.

For the benefit density functions of the jobs, we tried different non-negative, non-increasing functions such as $B(x) = \frac{1}{nx}$, $B(x) = \frac{n}{x}$ and so on; where x and n were both positive integer numbers.

We generated hundreds of job sets with randomly generated numbers as their arrival times and execution times, using the Poisson distribution which is applied by operations research to model random arrival times, especially for systems using queues, such as web servers and print servers [17].

1.4.1.1. Synthetic Job Sets

This experiment was done using aperiodic job sets. Each aperiodic job arrives with its own benefit density function and execution time. The arrival time and execution time of the jobs were randomly generated. Both algorithms were tested by scheduling hundreds of job sets with 20, 40, 60, 80, and 100 jobs which were randomly generated using a Poisson distribution. It is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event. This is the same attribute we have considered for the job sets, being independent and aperiodic. We used the range of [0, 10] for their arrival times and [1, 15] for their execution times. We simulated the scheduling of the job sets for the systems having 2 and 3 identical processors.

Also, we tested both algorithms by scheduling sets of 50, 100, 150, 200, and 250 jobs for the systems having 4 and 8 identical processors. Their arrival times range was [0, 10] and their execution times range was [1, 20]. We considered a shorter range for the arrival times comparing to the range of their execution times to simulate systems with heavier workloads comparing to the job sets generated for 2 and 3 processor systems.

1.4.1.2 Experimental Results and Discussion

The comparison of the results for BBA and LBBA are shown in separate graphs for each performance measurement and different number of processors. The improvement (or increase) in the total benefit gained by LBBA, ranged from 6.19 % to 9.0 % for a 2 processor system and from 5.04 % to 7.10 % for a 3 processor system (Figure 1.6).

The improvement (or increase) in the total benefit gained by LBBA ranged from 4 % to 16.73 % for a four-processor system and from 4.6 % to 16.35 % for a eight-processor system. As Figures 1.6 and 1.7 illustrate, the more jobs we had in the system, the greater was the improvement in maximizing benefits by LBBA compared to BBA.

The improvement (or decrease) of total processor idle time obtained by LBBA ranged from 44.21 % to 69.18 % for a 2 processor system, from 36.82 % to 70.68 % for a three-processor system (Figure1.4), from 39.18 % to 72.2 % for a four-processor system, and from 27.92 % to 57.83 % for eight-processors (Figure 1.5). These results prove that LBBA consumes CPU cycles more conservatively.

The tables contain the actual data gathered through our extensive experiments, which is not normalized. The arrival times and computation times are randomly generated and job sets can consist of both synchronous and asynchronous jobs. These factors affect the workload of the system, which itself has a direct impact on the total benefit gained, number of completed jobs and total processor idle time. We intentionally consider a variety of possible scenarios to study how both algorithms perform in each situation.

Despite of a little fluctuation, both BBA and LBBA have an increasing trend in total benefit as the number of jobs in the system increases. Having fewer jobs in a period of time and heavier workload during another period would result in more processor idle time (while fewer jobs are in the system) and less benefit gained by the completed jobs that had to wait longer in the system during the heavier period. This can be the reason of the fluctuation of the graph for 100 and 150 jobs. BBA has an increasing trend of total idle time with increasing number of jobs in the system, even with more processors. However, total processor idle time in LBBA stays almost the same and considerably less than BBA.

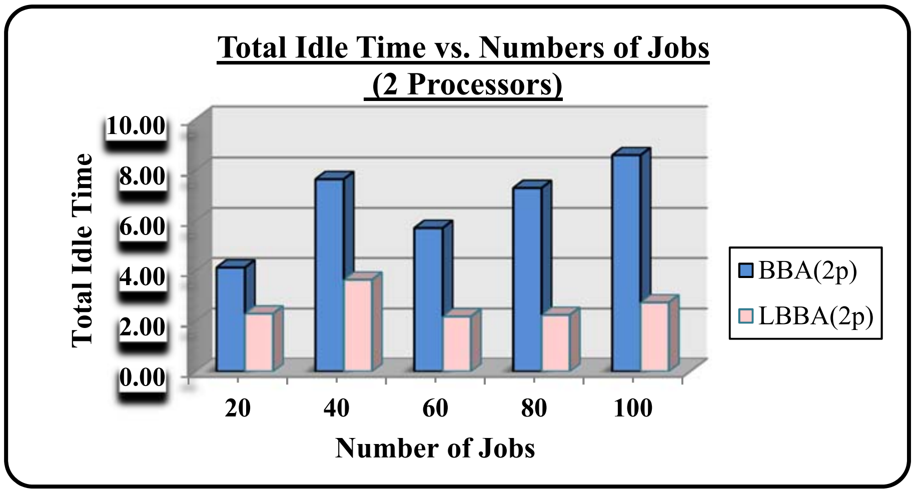
It also decreases in an 8 processor system and the difference of the trends is substantial in this case. Overall, LBBA reduced the total processor idle time by 24% (minimum) to 71% (maximum) compared to BBA.

The improvements in both total gained benefit and benefit-to-cost ratio (Tables 1.5 through 1.7) by the LBBA algorithm show that our new method can gain more benefit at lower cost, i.e., shorter makespan. Also, substantial decrease in total processor idle time by LBBA is a proof of its better resource management and CPU cycles usage.

However, one may argue that even though LBBA has much lower total idle time, it might have been using the processors to work on some jobs that are sent to garbage collection and their processing time is wasted not gaining any benefit while BBA was idle not starting those jobs. Our results (provided in Tables 1.8 and 1.9) prove that it is not the case.

We observed a significant improvement in the scheduling speed by our proposed method (Table 1.8). For example, the simulation of the previous method took 16 minutes (960 seconds) on average to schedule sets of 250 jobs on four-processors, while the average simulation time of our scheduling method was about 5 seconds for the same setting. The difference in the speed of the methods gets more significant for larger job sets. This is while the same design and implementation methods were used for both algorithms.

In addition, Table 1.9 provides the average ratio of missed deadlines for both algorithms. It shows that in all experimental settings, LBBA either had completed all the jobs, or if not, its miss ratio was lower than BBA for the same experiment. Therefore, we were able to gain more benefit in a shorter makespan, with less miss ratio and more balanced usage of processors resulting in lower total idle time.



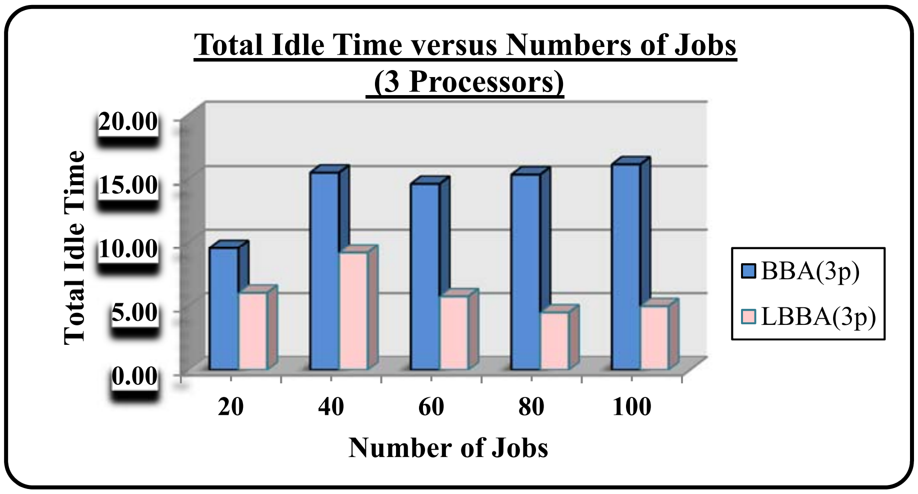
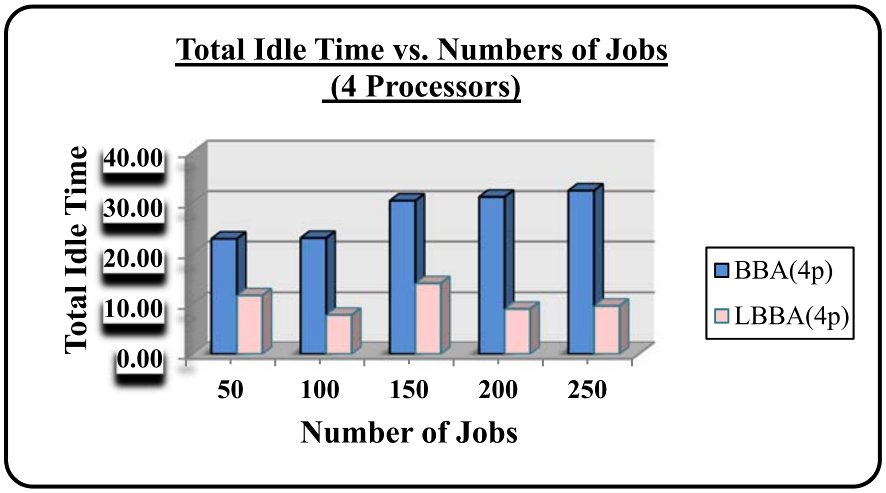


Fig. 1.4: Total idle time versus number of jobs for 2 and 3 processors
(Total idle time can be in milliseconds or microseconds. Time unit is not included in the chart, since the results are extracted from simulation experiments.)

Table 1.3: Total idle time and the improvement (decrease) achieved by LBBA on systems with 4 and 8 processors

No. of Jobs	BBA (2p)	LBBA (2p)	Improvement	BBA (3p)	LBBA (3p)	Improvement
20	4.13	2.30	**44.21%**	9.57	6.04	**36.82%**
40	7.65	3.65	**52.29%**	15.50	9.20	**40.65%**
60	5.71	2.19	**61.67%**	14.62	5.76	**60.59%**
80	7.30	2.25	**69.18%**	15.35	4.50	**70.68%**
100	8.60	2.75	**68.02%**	16.15	5.00	**69.04%**



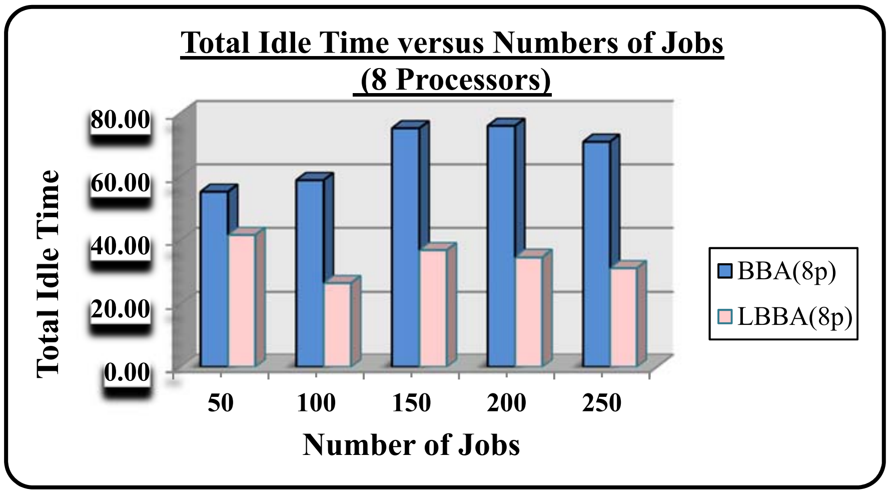
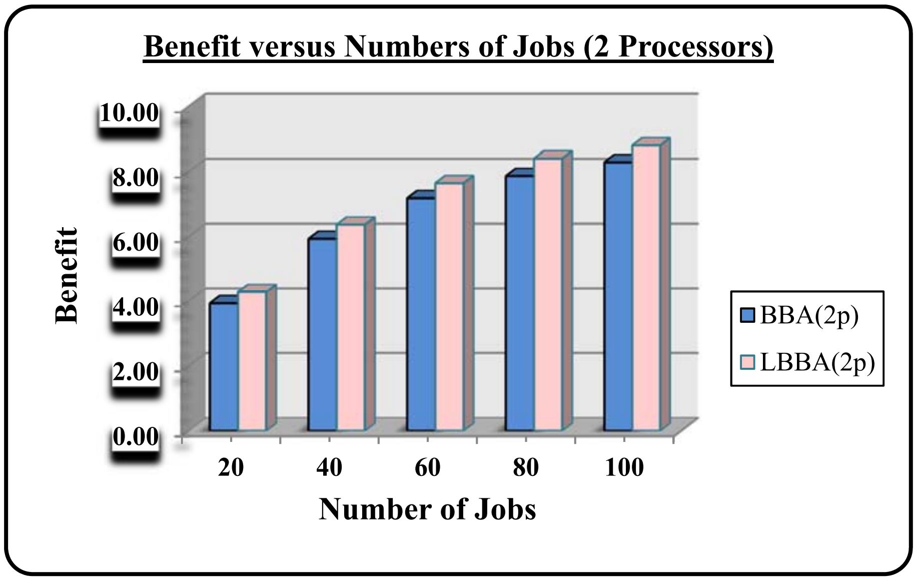


Fig. 1.5: Total idle time versus number of jobs for 4 and 8 processors *(Total idle time can be in milliseconds or microseconds. Time unit is not included in the chart, since the results are extracted from simulation experiments.)*

Table 1.4: Total idle time and the improvement (decrease) achieved by LBBA on systems with 4 and 8 processors

No. of Jobs	BBA (4p)	LBBA (4p)	Improvement	BBA (8p)	LBBA (8p)	Improvement
50	22.85	11.70	**24.59%**	55.30	41.70	**24.59%**
100	23.00	7.80	**66.09%**	58.95	26.70	**54.71%**
150	30.40	14.10	**53.62%**	75.30	36.90	**51.00%**
200	31.15	9.00	**71.11%**	76.05	34.80	**54.24%**
250	32.40	9.60	**70.37%**	71.05	31.40	**55.81%**



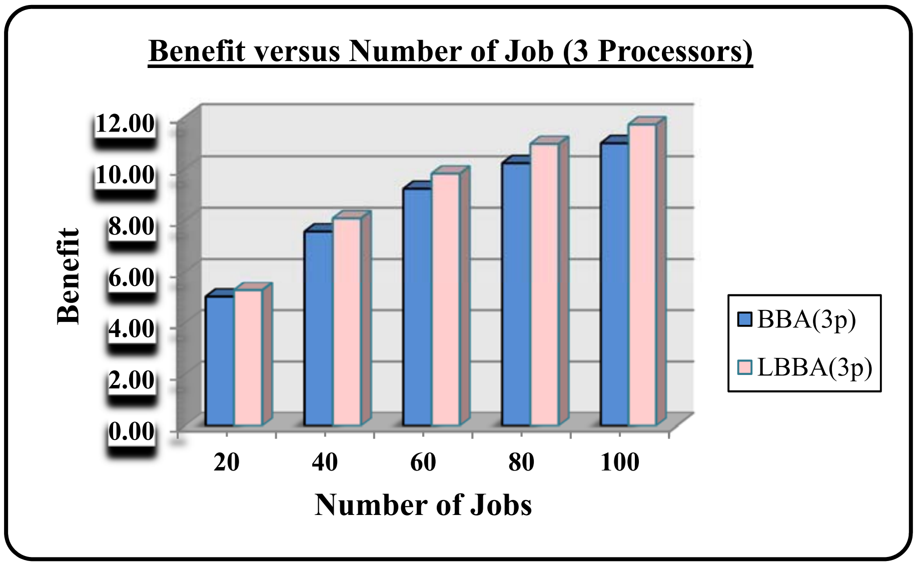
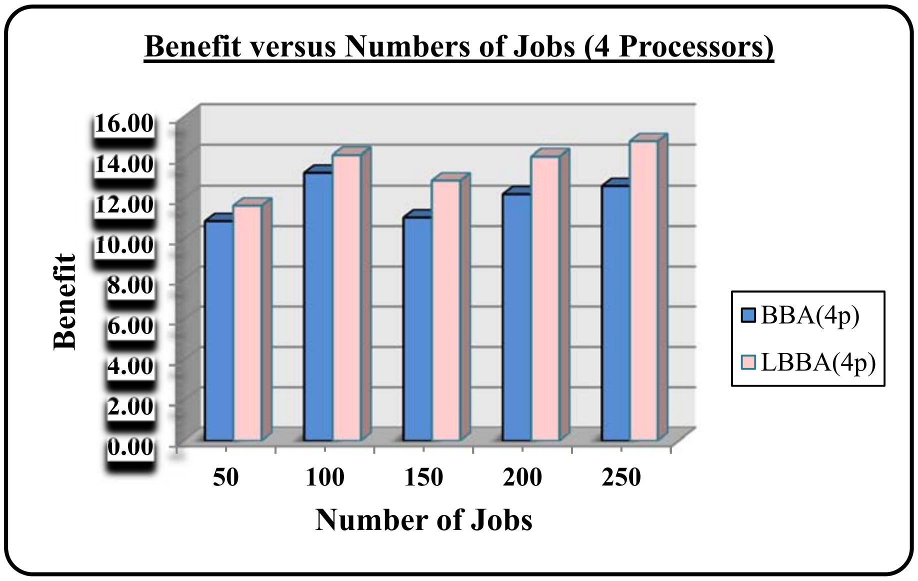


Fig. 1.6: Benefit versus number of jobs for 2 and 3 processors

Table 1.5: Benefit gained by BBA and LBBA, and the improvement (increase) achieved by LBBA on systems with 2 and 3 processors

No. of Jobs	BBA (2p)	LBBA (2p)	Improvement	BBA (3p)	LBBA (3p)	Improvement
20	3.92	4.28	**9.18 %**	5.02	5.27	**4.98 %**
40	5.92	6.36	**7.43 %**	7.57	8.08	**6.74 %**
60	7.18	7.64	**6.40 %**	9.23	9.82	**6.39 %**
80	7.86	8.40	**6.87 %**	10.23	10.95	**7.04 %**
100	8.29	8.80	**6.15 %**	10.98	11.71	**6.65 %**



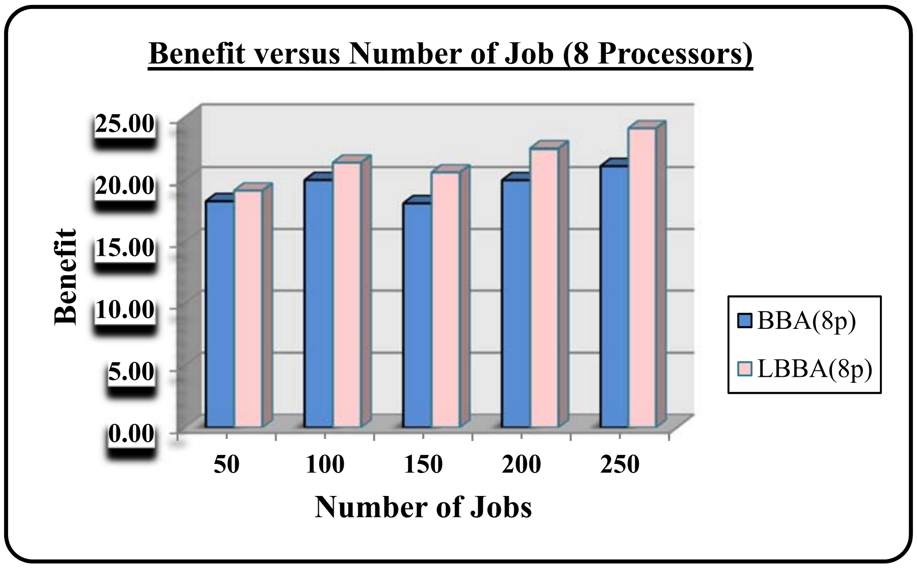
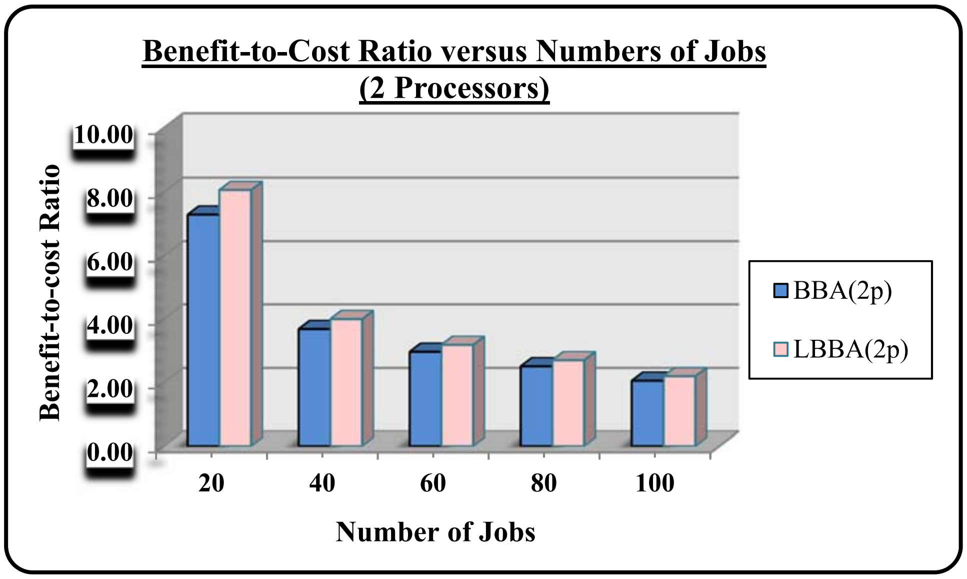


Fig. 1.7: Benefit versus number of jobs for 4 and 8 processors

Table 1.6: Benefit gained by BBA and LBBA, and the improvement (increase) achieved by LBBA on systems with 4 and 8 processors

No. of Jobs	BBA (4p)	LBBA (4p)	Improvement	BBA (8p)	LBBA (8p)	Improvement
50	10.88	11.65	**7.08%**	18.27	19.13	**4.65%**
100	13.27	14.14	**6.56%**	19.98	21.38	**7.01%**
150	11.06	12.89	**16.55%**	18.11	20.59	**13.69%**
200	12.22	14.07	**15.14%**	19.97	22.48	**12.57%**
250	12.62	14.81	**17.35%**	21.11	24.10	**14.16%**



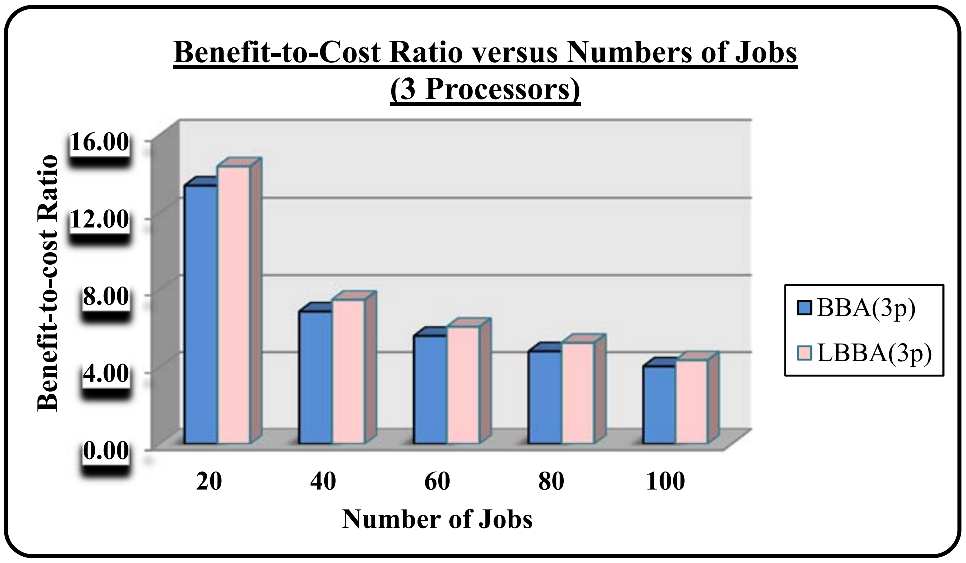
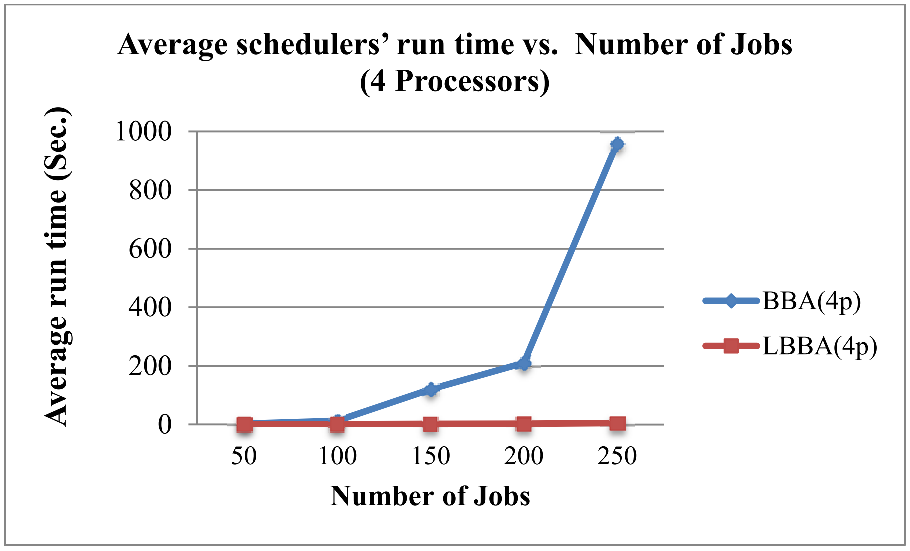


Fig. 1.8: Benefit-to-Cost ratio versus number of jobs for 2 and 3 processors

Table 1.7: Benefit-to-cost ratio and the improvement achieved by LBBA on systems with 2 and 3 processors

No. of Jobs	BBA (2p)	LBBA (2p)	Improvement	BBA (3p)	LBBA (3p)	Improvement
20	7.29	8.06	10.62%	13.40	14.40	7.45%
40	3.68	3.99	8.53%	6.87	7.47	8.77%
60	2.96	3.18	7.32%	5.60	6.08	8.40%
80	2.50	2.69	7.67%	4.81	5.25	9.13%
100	2.05	2.19	6.86%	4.03	4.35	7.91%



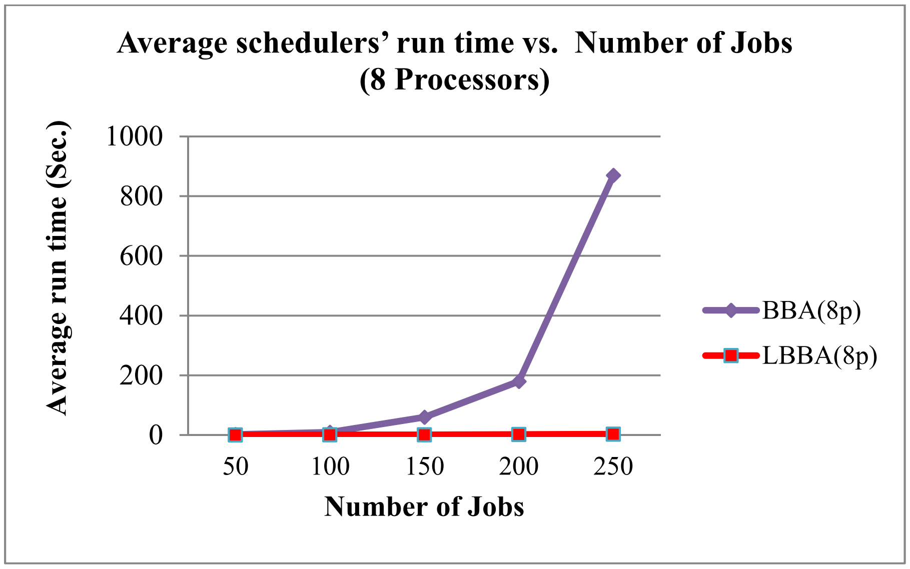


Fig. 1.9: Average schedulers' run time vs. number of jobs for 4 and 8 processors

Table 1.8: Average schedulers' run time (in seconds)

Number of Jobs	BBA (4p)	LBBA (4p)	BBA (8p)	LBBA (8p)
50	2.50	1.00	2.00	0.50
100	12.50	1.00	9.60	1.00
150	120.00	2.00	60.00	1.00
200	210.00	3.00	180.00	2.00
250	960.00	5.00	870.00	3.00

Table 1.9: Missed deadline ratio (%)

No. of Jobs	BBA (4p)	LBBA (4p)	BBA (8p)	LBBA (8p)
50	0.084	0.00	0.00	0.00
100	0.100	0.05	0.00	0.00
150	0.067	0.00	0.00	0.00
200	0.075	0.00	0.05	0.00
250	0.100	0.00	0.02	0.00

1.5. Summary

In this chapter, I proposed and analyzed an efficient preemptive benefit-aware technique that prioritizes the jobs using their benefit density functions and schedules them in a real-time basis in order to maximize the total benefit gained by the completed jobs. It reduces the cost (makespan or overall flow time) of the jobs and total idle time of the processors by online choice of greedy or load balancing approximations to partition jobs among multiple processors at the time of release.

I proposed an online choice of two approximation algorithms (greedy and load-balancing), as a solution for special cases that were not considered in the existing benefit-aware multiprocessor scheduling algorithms. Examples of those cases are when there are several jobs with the same priority or when a high priority job can be executed by more than one non-idle processor because it can preempt the running jobs on the top of their stacks.

The results of the theoretical analysis and simulation experiments show that LBBA has improved the performance of the previous benefit-based method (BBA) explained in [29] while preserving its constant competitive ratio ($10C^2$). Our extensive experiments showed up to 72% improvement in total processor idle time, and improving the benefit accrual by up to 17% compared to BBA. LBBA showed more improvement with heavier workload. This improvement is provided by more balanced distribution of jobs among processors and shorter flow times which will increase the total benefit. Also, a better resource management (CPU cycles in this case) is possible using the proposed method. This advantage is beneficial to many real-time applications especially those that are running on battery-operated computing devices.

In addition, the results of the theoretical analysis and simulation showed that this solution is computationally less expensive so that the LBBA scheduling algorithm worked up to about 290 times faster than BBA (3 seconds for scheduling 250 jobs on an eight-processor system compared to 870 seconds taken by BBA). This advantage along with a lower ratio of missed deadlines makes LBBA a considerably faster scheduling algorithm for multiprocessor systems comparing to the previous algorithm (BBA) which can improve the QoS (Quality of Service) for many real-time applications such as bursty hosting servers, video games and other multimedia systems. Another advantage of this new method is that no synchronization is needed for fetching the jobs from a shared pool. That is because a separate pool is assigned to each processor in contrast to the other method (BBA) where all processors use a shared pool.

We can summarize the advantages of LBBA as follows:

- **LBBA is a novel hybrid model** of soft real-time multiprocessor scheduling. In contrast to BBA, which only follows a benefit model, LBBA is a combination of benefit and cost models. That is, it aims to **minimize makespan** in order to achieve the *maximum benefit at the lowest cost*.

- **No synchronization is needed** for fetching the jobs from a shared pool. That is because a separate pool is assigned to each processor in contrast to the other method (BBA) where all processors use a shared pool.

- **LBBA facilitates load-balanced partitioning** of waiting jobs, while this case is not considered in BBA. For example, in case the waiting (or ready) jobs arrive *asynchronously*, LBBA adapts the "greedy approximation" to assign a job to the pool of the processor with the least remaining workload. If jobs are *synchronous*, i.e., arrive at the same time, those that cannot start running and have to wait in a pool, will be partitioned among the processors, using our "Load-Balancing" technique.

- **LBBA optimizes the CPU usage and minimizes the total idle time** of the processors by balancing the workload among them.

- **Improved QoS by reducing missed deadline ratio:** As shown by an example in the manuscript, LBBA reduces the possibility of starvation for low priority jobs, compared to BBA.

- **LBBA is computationally less expensive** than BBA, as proved in this chapter.

- **Minimal Response time:** The overall response time, including both scheduling and execution time, for a job set in LBBA is much shorter than in BBA for the same set. The heavier the workload of the system, the greater the reduction of response time (up to about 300 times faster for a set of 250 jobs).

In the next phase of my research, I extended this work to the problem of scheduling periodic task sets, proposed other versions of LBBA to enhance the QoS in applications with periodic tasks, and explored their impacts on the benefit, cost, and schedulability in those cases.

2. Online Benefit-Aware Semi-Partitioned Scheduling of SRT Periodic Tasks

This chapter presents a novel benefit-aware semi-partitioning approach to enhance scheduling of soft real-time periodic tasks in multiprocessor systems. Tasks in these systems are not equally critical (or beneficial). Each task comes with its own benefit density function which can be different from the others'. Examples of such applications and embedded systems can be body sensor networks and real-time medical monitoring systems which periodically check and record patients' vital signs, video-streaming servers, multi-player video games, and mobile multimedia devices. The tasks are prioritized based on their potential benefits for the system. The sooner a task completes the more benefit it gains. The objective is enhancing the QoS by increasing the total benefit, while reducing flow times and deadline misses. Theoretical performance analysis of this model is provided, followed by two more versions of the algorithm, one for tasks with non-implicit deadlines, and the other with a modified load-balancing factor. A general framework for benefit-aware multiprocessor scheduling in applications with periodic, aperiodic or mixed real-time tasks is also proposed in this work. The extensive simulation experiments compare these algorithms with the state-of-the-art, using different utilization levels and various benefit density functions. The results of these comparisons show that the new techniques outperform the existing ones.

2.1. Introduction and Motivation

2.1.1. Background

Multiprocessor systems are widely used in a fast-growing number of real-time applications as well as embedded systems. In hard real-time systems, meeting all deadlines is critical, while in soft real-time (SRT) systems, missing a few deadlines does not drastically affect the system performance. However, it would compromise the quality of the service (QoS). Some examples of such systems are video conference applications, online gaming, e-commerce transactions, chatting, IM (instant messaging), Cloud applications [33] and IoT (Internet-of-Things) [16].

In such systems, jobs meeting their deadlines will gain a *benefit* (also called *reward*) for the system. Hence, researchers focus on increasing benefits to improve the QoS. Besides the total benefit, other factors also influence the QoS, such as *makespan* (the time difference between the start and finish of a sequence of jobs or tasks), *flow time* (the time needed to finish a job), and the deadline-miss ratio. However, most existing scheduling models focus on one of these factors in order to improve the system performance. The major approaches, which multiprocessor real-time scheduling algorithms follow, are partitioning, global scheduling, and some hybrid of the two, called semi-partitioning [14], [17], and [19].

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 [33]. These issues are discussed at some length by Srinivasan *et al.* [34].

2.1.2. Related Works

We discuss the related works in the following categories:

2.1.2.1. QoS and Benefit-aware Scheduling

Elnably *et al.* [33] study fair resource allocation and propose a benefit-aware model for QoS in Cloud applications. In contrast, Alhussian, Zakaria and Hussin [35] 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 [36] have discussed specification and management of QoS in real-time databases supporting imprecise computations. Benefit-aware scheduling of periodic tasks on uniprocessor systems has also been studied by Aydin *et al.* [30], and Hou and Kumar [37]. Zu and Cheng [38] proposed a real-time scheduling method for tasks, with hierarchically dependent benefit-aware sub-tasks, through a multimedia and image/video transmission case study. Chen, Kuo and Yang discussed a profit-driven uniprocessor scheduling with energy and timing constraints [39]. Awerbuch *et al.* [29] proposed a benefit-aware model for scheduling aperiodic tasks on uniprocessor systems which can also be applied to multiprocessors.

We have also studied benefit-aware scheduling of aperiodic real-time tasks on multi-processor systems (Sanati and Cheng in [20], [21], and [32]). The performance analysis and comparative experimental results of our proposed algorithms versus another state-of-the art method proposed in [29] showed that our technique achieved significant improvements in reducing the overall response time (i.e., scheduling time plus makespan of the task sets), increasing the total benefit and reducing missed deadlines, all of which enhance QoS. However, that method is designed for scheduling one instance (i.e., aperiodic) tasks only, and cannot solve the problem of scheduling periodic soft real-time tasks on multiprocessor systems, on which relatively very little research has been done.

2.1.2.2. Semi-Partitioned Scheduling

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 to 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 [40] 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 (Earliest-Deadline-First) scheduling are used. In EDF-fm, the number of processors to which jobs of a migrating task can migrate to, is limited to two, and in EDF-os, each processor can be assigned to only two migrating tasks. The goal in these EDF-based semi-partitioning strategies is to minimize tardiness.

2.1.2.3. Approximation Algorithms in Scheduling

Approximation algorithms are often used to attack difficult optimization problems, such as job scheduling on multiprocessor systems which is an NP-hard problem [1]. We applied greedy and load-balancing algorithms for benefit-aware multiprocessor scheduling of aperiodic real-time tasks [20], [21], [32]. Chen, Yang and Kuo [19] used greedy approximation in real-time task replication for fault tolerance in identical multiprocessor systems. Chen and Chakraborty [41] have studied the approximation of partitioned scheduling by exploiting resource augmentation with (1) speeding up or (2) allocating more processors.

2.1.3. The Objective

Our objective in this study is to enhance the QoS by reducing flow times and missed deadlines, while increasing the total benefit obtained by completed periodic tasks. Hence, we semi-partition the tasks and allow different jobs of any task to be assigned to different processors as they arrive (migration at job boundaries) based on their benefit-aware priorities and their workloads. 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.

2.1.4. The Contribution

In this work, we propose a new technique which, to our knowledge, is the first *online benefit-aware semi-partitioned scheduling for 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. As in LBBA (load-balanced benefit-aware algorithm) [21], we use an online choice of two approximation algorithms (load-balancing and greedy approximation) for partitioning lower priority tasks that are waiting, at job boundaries and no migration is allowed after a job (or sub-task) is assigned to a processor. However, unlike the original LBBA, the method proposed in this work is designed for periodic tasks and works for systems with both implicit and non-implicit deadlines.

We summarize some highlights of this technique as follows:

- An enhanced usage of the processing time by approximately balancing the workload of the processors, which reduces the idle times and flow times
- When different benefit density functions are assigned to different tasks in a system, it increases the total gained benefit by prioritizing tasks based on their benefit density functions.

Our 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, and no need to pre-select migrating tasks (different jobs of which can be assigned to different processors).
- In EDF-fm, the number of processors on which different jobs of each migrating task can be processed is limited to two. In EDF-fm and EDF-os, each processor cannot

accept jobs from more than two migrating tasks. Our proposed method has neither of these limitations.

- It reduces runtime overhead by not allowing migration in the middle of job executions (which is allowed in global scheduling methods such as Global EDF [43]).
- It also reduces overhead by keeping the preempted jobs and the running one on the same stack for each processor (details in Section 2.2), unlike EDF-os which replaces the preempted jobs by the running ones, appends the preempted jobs to a queue and fetches them again at their resume time.

In the next section, we explain our novel semi-partitioning hybrid model called LBBA-bid, which combines benefit and cost models, for optimizing QoS in soft real-time systems of periodic tasks with benefit-aware implicit deadlines. In Section 2.3, we provide the theoretical analysis of this algorithm. Sections 2.4 and 2.5 respectively include our proposal of two more variations of LBBA-bid, one with a non-implicit deadline definition (LBBA-bnc) and the other one with a different factor considered for load-balancing (UBBA). In Section 2.6, we demonstrate all three proposed models through an example. Section 2.7 includes the performance analysis of all three proposed approaches based on the results of our extensive simulation experiments on synthetic task sets in comparison with the state-of-the-art. In Section 2.8, we conclude this work. and suggest the future work. Based on our conclusion, we introduce a novel general framework for benefit-aware multiprocessor scheduling of SRT systems with aperiodic, periodic or mixed tasks, performance analysis of which can be a worthwhile future work.

2.2. LBBA-bid for Implicit Deadlines

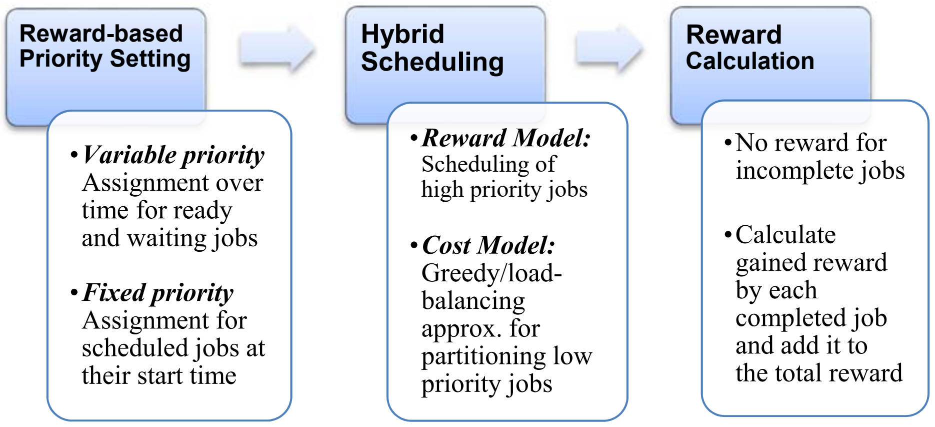
In this section, I define the system and task model, methodology and notations/phrases used in our proposed LBBA-bid algorithm [44] for periodic tasks with implicit deadlines.

2.2.1. 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. Preemption is allowed. A desired property of the system in this method is the possibility to delay jobs without drastically reducing the overall system performance.

2.2.2. Methodology

A hybrid model (combining benefit and cost models) is proposed 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. Figure 2.1 summarizes our hybrid methodology for scheduling 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:
Scheduling of high priority jobs
Cost Model:
Greedy/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. 2.1: Our method

2.2.2.1. Semi-Partitioning Model

This algorithm applies online semi-partitioning. In this 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 (with no migrations), based on the job's priority, worst-case execution time, and the current workloads of the processors. 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 runtime overhead of job migrations which is allowed in global scheduling.

2.2.2.2. Online Choice of Approximation Algorithms

Similar to LBBA, we consider greedy and load-balancing approximation algorithms, one of which will be chosen online based on the conditions of the system to distribute the waiting tasks among the processors at job boundaries and/or schedule higher priority tasks whenever there are several possible choices. Flow time reduction, and also enhanced CPU usage by reducing idle times are advantages of this technique.

2.2.3. Definitions

The definitions of the phrases and notations used in this method are as follows:

2.2.3.1. 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 is known as a job and is denoted as $T_{i,j}$, 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 task T_i, which is $T_{i,j}$, would be the arrival time of job $T_{i,(j+1)}$, that is at jp_i. However, it can be *non-implicit* and defined based on objectives and criticalities of the systems and applications. Time is slotted and expressed by t. To demonstrate that our model can be used to model a video streaming server, a frame can be considered as the time between two consecutive time slots, where all tasks generate a job and the length of a frame is the least common multiple of task periods [37].

2.2.3.2. Task Utilization

Another important parameter used to describe a task T_i 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.

2.2.3.3. *Notation*

The notation used throughout this chapter is defined 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

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

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

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

$Br_{i,j}$ – *benefit-aware break point* of job $T_{i,j}$, 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, even if its deadline has not passed.

Note: The formulas for this benefit-aware break point, benefit density function and its restriction, gained benefit, variable and fixed priority of a job, as provided below, have been originally proposed (with explanation and analysis) by Awerbuch et al., in [29], and were also adopted in LBBA [21], [32].

$\beta_i(t)$ – *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 $\beta_i(t)$:

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

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. The above condition guarantees 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. Constant C and the benefit-density functions are defined based on the requirements of real applications and can be different from one application to another.

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

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

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

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

β_i is a non-negative non-increasing function; thus, 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 $T_{i,j}$ 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 $T_{i,j}$, 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 $T_{i,j}$,

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):

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

Wp_l – *Current execution time* on the pool of processor l

Ws_l – *Current execution time* on the stack of processor l

W_l – *Current workload or execution time* on processor l:

$$W_l = Wp_l + Ws_l$$

U_l, Up_l, Us_l – *Total current utilization* on processor l, its pool and stack, respectively.

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

$$U = m$$

u_i – *Utilization* of every job of the task T_i :

$$u_i = w_i / p_i$$

$£_i$ – *Laxity* of job $T_{i,j}$:

$$£_i = p_i - w_i \tag{4}$$

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

$$\boldsymbol{\delta}_{i,j} = s_{i,j} - r_{i,j} \tag{5}$$

$\boldsymbol{\varphi}_{i,j}(k)$ – *time elapsed during the* k^th^ *preemption* of $T_{i,j}$

$\boldsymbol{\varphi}_{i,j}$ – *time elapsed during all the preemptions* of $T_{i,j}$

$\boldsymbol{\S}_{i,j}$ – *stretch time of* job $T_{i,j}$, that is the extra time $T_{i,j}$ stays in the system in addition to its execution time.

$$\S_{i,j} = f_{i,j} - w_i \tag{6}$$

$$\S_{i,j} = \boldsymbol{\delta}_{i,j} + \boldsymbol{\varphi}_{i,j} \tag{7}$$

2.2.4. LBBA-bid 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. Arriving jobs are prioritized, and then are either scheduled and started to run 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. This algorithm is called *LBBA-bid*, *LBBA with **b**enefit-aware **i**mplicit **d**eadlines.*

The algorithm consists of the following phases:

2.2.4.1. Prioritizing

The priority of each ready and unscheduled job (located in each pool) at time *t,* 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 pair of *i,j*) starts its execution, its priority is calculated as $d'_k = \beta_k (s_k + w_k - r_k)$ (lines 12 and 46 of the pseudo-code, Algorithm 2). 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*, thus 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 $t > s_k$).

2.2.4.2. Scheduling / Execution / Preemption

Once a new job $T_{i,j}$ is released, if there is a processor such that its stack is empty (lines 9 through 20), then the newly released job is pushed onto the stack and starts running. If there is no idle processor, but the preemption condition (line 52) is met for any running job, then the job $T_{i,j}$ preempts the one currently running, and starts its execution. As mentioned in Sub-section 2.3.3, the preemption condition we applied here, was originally proposed by Awerbuch et al. [29]. They used this condition to prove the constant ratio competitiveness of $10C^2$ for their benefit-aware algorithm (Constant C is defined in 2.3.3). It also limits the number of preemptions and their overhead by preventing a new job $T_{i,j}$ from preempting the running jobs with lower priorities unless the priority of the new job is more than four times the priority of running job(s).

2.2.4.3. 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 job with the largest execution time, w, to the processor with the minimum remaining workload, the second largest job to the processor with the second smallest remaining workload, so on so forth. This way we are able to balance the workload among the processors (lines 24 through 36).

However, in case there is only one high priority job at a time instant which can preempt more than one running job, we assign it to the stack of the processor with minimum remaining execution time (greedy approximation, lines 42 through 49). 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.

2.2.4.4. Reaching Break Point or Deadline

If a job reaches its break point or its deadline (lines 50-56) 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 garbage collection. The deadline of a job is its period (Next-Job-Release time of the same task) and its break point [29] 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.

2.2.4.5. 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 $T_{i,j}$ 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, its gained benefit is calculated and added to the total benefit (line 62).

The summary of the algorithm is provided in pseudo-code as follows:

ALGORITHM 2: LBBA-bid

```
Required: One or more jobs arrive at time $t \geq 0$
{
    /* TempList: list of ready jobs waiting for distribution among processors */
    Append the arrived job(s) to the TempList
    Calculate the priority of each job $T_{i,j}$ in the TempList: $d_{i,j}(t) = B_i(t + w_i - r_{i,j})$
    Sort TempList based on the priority
    If (at least one stack is empty)
    {
        Push the highest priority job(s) $T_{i,j}$ onto empty stack(s) of idle processor(s) $l$
        Add its execution time $w_i$ to total workload of the stack of the processor $l$ ($Ws_l$)
        Recalculate total workload of processor $l$: $W_l = Wp_l + Ws_l$
        Calculate the fixed priority of $T_{i,j}$ using its start time $s_{i,j}$: : $d'_{i,j}(t) = B_i(s_{i,j} + w_i - r_{i,j})$
        Start executing $T_{i,j}$
    }
    Else
    {
        /* no empty stack */
        /* preempt if possible otherwise partition 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 ($d_{i,j}(t) \leq 4d'_k$ (for each job $T_{i,j}$ in TempList and each running job $T_k$))
        {
            /* no preemption allowed */
            /* partition the ready jobs among the pools (Load-Balanced Partitioning) */
                For (each job $T_{i,j}$ in TempList)
```

ALGORITHM 2: LBBA-bid (Continued)

```
            {
                **Sort** the processors in ascending order of their total remaining workload on
                        their pools and stacks:
                        $W_l = Wp_l + Ws_l$
                **Append** the job $T_{i,j}$ with largest execution time $w_i$ to the pool of the processor $l$
                        with the minimum remaining work load;
                /* *load balancing* */
                **Remove** $T_{i,j}$ from TempList
                **Add** its execution time $w_i$ to total workload of the pool of processor $l$ ( $Wp_l$ )
                **Recalculate** total workload of processor $l$:   $W_l = Wp_l + Ws_l$
            }
        }
    **Else**
    /* if ($d_{i,j}(t) > 4d'_k$)  then ( $T_{i,j}$ preempts $T_k$)*/
    /* If $T_{i,j}$ has more than one choice of processors, it will be pushed onto the stack whose
        processor has the least workload (*greedy approximation*) */
    {
        **Stop** the execution of job $T_k$ (preempt $T_k$)
        **Push** the job $T_{i,j}$ onto the stack on top of $T_k$
        **Start** executing $T_{i,j}$
        **Calculate** the fixed priority of $T_{i,j}$ using $s_{i,j}$,:  $d'_{i,j}(t) = B_i(s_{i,j} + w_i - r_{i,j})$
        **Add** the execution time of $T_{i,j}$ to the total workload of that stack ($Ws_l$ )
        **Recalculate** total workload of the Processor $l$:  $W_l = Wp_l + Ws_l$
    }
    /* check if any of the running jobs on top of the stacks has reached its
```

ALGORITHM 2: LBBA-bid (Continued)

```
51            deadline $D_{i,j}$ ($D_{i,j} = r_{i,j} + p_i$) or break point $Br_{i,j} = s_{i,j} + 2w_i$:*/
52        **If** $t > \min(D_{i,j}, Br_{i,j})$
53        {
54            **Remove** the job from the stack
55            **Send** it to the Garbage Collection of the processor;
56        }
57        **Else** (if $T_{i,j}$ not preempted), continue its execution
58        /*  for every completed job $T_{i,j}$ calculate benefit $b_i$ */
59        $b_i = w_i . \beta_i(f_i)$
60    }
61    /*  calculate the sum of all benefits gained, $q$ being the number of completed jobs */
62    $B = \sum_{j=1}^{q} bj$
63    }
```

2.3. Analysis

2.3.1. 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 (more details and analysis provided in Chapter 1); otherwise, it would be discarded from the system without gaining any benefit. This means no job in this method can be tardy, i.e., finish its execution after its deadline. Therefore, having tardiness equal to zero, there is no need for a tardiness bound analysis. However, the above restriction will induce an upper bound on the delay each job may have after being released till it is scheduled and starts its execution.

Theorem 1 – *If $Br_{i,j} > D_{i,j}$, and $T_{i,j}$ is not preempted while running, the latest time $T_{i,j}$ can be started, while remaining schedulable is $Max\ (s_{i,j}) = r_{i,j} + £_i$.*

Proof. Recall 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:

$$Max\ (s_{i,j}) = r_{i,j} + £_i \qquad \blacksquare$$

Corollary 3.1.1 – *Job Schedulability Condition: If a job $T_{i,j}$ is scheduled at its $Max(s_{i,j})$ and $Br_{i,j} > D_{i,j}$, 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 $T_{i,j}$ is preempted, then $T_{i,j}$ will miss the deadline and will not gain any benefit.*

Proof. From Theorem 1, we deduce that $T_{i,j}$ must complete its execution without any interruption to be schedulable, because the time period between its start time and deadline is exactly equal to its execution time. Thus, if it is preempted, it will not be able to meet its deadline. ■

Theorem 2 – *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.*

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

$$s_{i,j} + 2w_i \leq D_{i,j}$$

$$s_{i,j} + 2w_i \leq r_{i,j} + p_i$$

$$s_{i,j} - r_{i,j} \leq p_j - 2w_i$$

$$\boldsymbol{\delta}_{i,j} \leq p_i - 2w_i \tag{8}$$

If $u_i \geq \frac{1}{2}$, $\quad p_i \leq 2w_i,$

and from (8): $\quad Max\,(\boldsymbol{\delta}_{i,j}) = 0$

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):

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

and from (6) and (7):

$$w_i + \boldsymbol{\varphi}_{i,j} + \boldsymbol{\delta}_{i,j} \leq p_i \tag{9}$$

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. (9))*:

$$Max\ (\varphi_{i,j}) = £_i$$

Proof. This is deduced from Theorem 2 and eq. (9). ■

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 with a waiting time (delay) cannot have a 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.

I proceed by proposing two modified versions of LBBA-bid in the Sections 4 and 5; the first is to be applied for systems with non-implicit deadlines, and the second is for balancing job utilization among processors instead of balancing their worst case execution time. Experimental performance evaluation was conducted for all three versions versus the state-of-the-art and is presented in Section 2.7.

2.4. LBBA-bnc for Non-Implicit Deadlines

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 it to ***b**enefit-aware non-implicit deadline of **n**ext-job-**c**ompletion time (bnc).* This non-implicit deadline definition is applicable and beneficiary to the applications or embedded systems in which the QoS expectation allows this relaxation of deadline. In LBBA-bnc, the definition of $D_{i,j}$ used in the LBBA-bid will be modified to:

Line 51: $D_{i,j} = c_{i,(j+1)}$

Then, if $T_{i,j}$ is still running on one processor when $T_{i,j+1}$ is released, there will be two possible cases.

Case (1) - The priority of $T_{i,j+1}$ is not high enough and it has to be partitioned and sent to a pool. Then, there will be two scenarios in which both jobs meet their deadlines and gain benefit for the system:

1. If $T_{i,j+1}$ is not sent to the pool of 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.
2. If $T_{i,j+1}$ is waiting on the pool of the same processor $T_{i,j}$ is running on, then $T_{i,j}$ has to complete before the laxity of $T_{i,j+1}$ ends. In this case, both jobs meet their deadlines.

Case (2) – The priority of $T_{i,j+1}$ is high enough to be scheduled on an idle processor or preempt a running job and starts its execution. Then, if $T_{i,j}$ completes while $T_{i,j+1}$ is still in the system, either running or preempted, $T_{i,j}$ meets its deadline and adds its gained benefit to the total benefit.

2.5. UBBA – Utilization Balancing

Some EDF-based semi-partitioning algorithms (e.g., EDF-os) consider balancing 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. However, LBBA-bid and LBBA-bnc balance the execution time among the processors. To be able to study the difference in the performance and QoS of the proposed methods and other (EDF-based) algorithms, I modified LBBA-bid to introduce another version of the algorithm, called UBBA, which partitions the jobs among the processors by approximately balancing their utilizations.

In the *UBBA* (***u****tilization-****b****alanced* ***b****enefit-****a****ware*) algorithm, I replaced the load-balancing part (lines 25-31) of the algorithm with the following lines and the same method applies to the greedy approximation. 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*). As shown in an example (Section 2.6) and in experimental evaluations (Section7), this method cannot balance the workload of the processors as well as LBBA-bid.

Utilization-Balancing Approximation *(for Partitioning)*

```
25 For (each job $T_{i,j}$ in TempList)
26 {
27         Sort the processors in ascending order of their
28                 total remaining utilization on their pools and
29                 stacks:
30                 $U_l = \sum Up_l + \sum Us_l$      // $l$ is processor index
31         Append the job $T_{i,j}$ with largest utilization $u_{i,j}$ to the pool…
```

2.6. A Motivating Example

This section demonstrates 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 Table 2.1. To simplify the example and make it easier to follow, assume that the tasks are synchronous and released at time $t = 0$, with the same benefit density function (e.g., $f(x) = 1/x$).

The tasks can represent the processing steps for different parts of a JPEG image handled by a real-time priority-driven coding and transmission scheme [42]. In this scheme, important parts of an image are given higher priority (and then higher benefits if completely processed) over less important parts. Thus, the high-priority parts can achieve high image quality, while the low-priority parts, with a slight sacrifice of quality, can achieve a significant compression rate and hence save the power/energy of a low-power wireless system.

The LCM (Least common multiple) of periods of these tasks (i.e., the hyper-period) is 30. Therefore, we illustrate the schedule within the first 30 units of time. During this time interval, 6 instances of T_1, 10 instances of T_2 and 3 instances of T_3 will be released. Their total utilizations will be $\frac{59}{30}$ ($\frac{18}{30} + \frac{20}{30} + \frac{21}{30}$). This is less than 2, i.e., the maximum possible utilization of a 2 processor system.

Table 2.1: An example of 3 periodic tasks

	T_1	T_2	T_3
W	3	2	7
P	5	3	10

Therefore, the necessary condition for the task set to be schedulable is met, although it would not be sufficient. The following sub-sections explain how our methods schedule the above task set in comparison with EDF-based schedulers, Global EDF and EDF-os.

2.6.1. Benefit-aware schedulers

We start the scheduling process using LBBA-bid and then explain how differently LBBA-bnc and UBBA schedule the same task set. The initial priority setting is the same in the three proposed method.

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_2) 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; 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.

The current remaining workload of each processor is as follows:

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

$W_2 = 1$ (remaining 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 $T_{2,4}$ ($w_{24}/f_{24} = 2/(11-9) = 2/2 = 1$) 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.

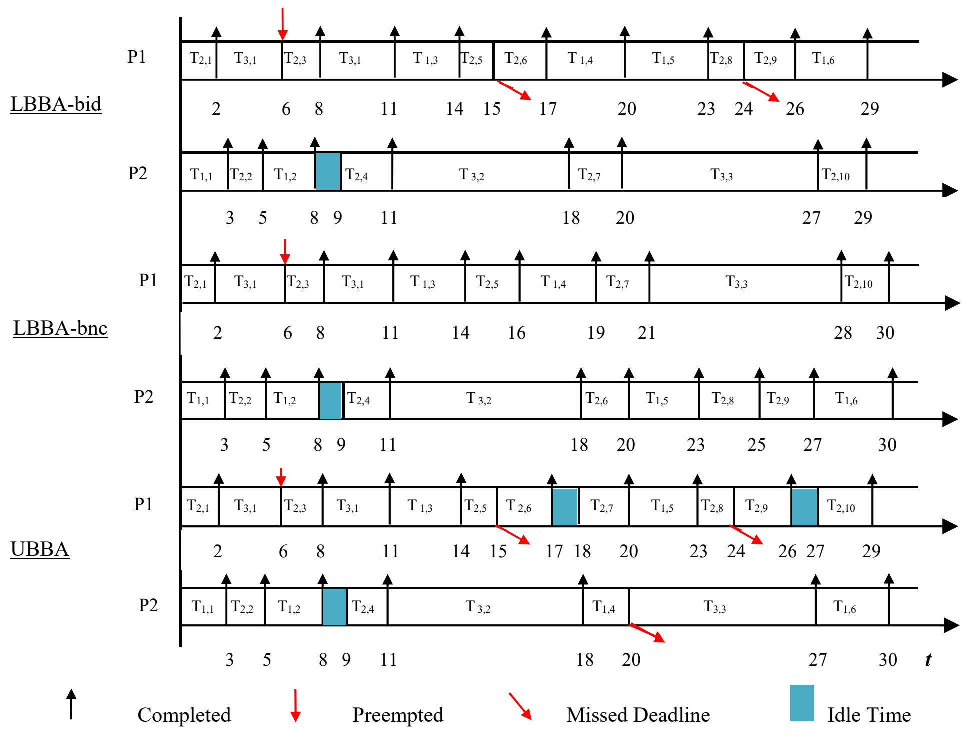


Fig. 2.2: Scheduling diagram for LBBA-bid, LBBA-bnc and UBBA

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 $T_{3,2}$)

The rest of the scheduling process is illustrated in Figure 2.2, along with the schedules provided by LBBA-bid and UBBA. 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.

In LBBA-bnc, priority settings, scheduling 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 yet completed. This occurs 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).

Hence, 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. Thus, this relaxation of the deadline would reduce the tardiness (i.e., the number of missed deadlines) and as shown in Figure 2.2, LBBA-bnc schedules all the jobs in this example, while two jobs in LBBA-bid and three jobs in UBBA method miss their deadlines.

The deadline in both LBBA-bid and UBBA is implicit, and both prioritize the tasks based on their benefit and approximately balance the processors workload. However, the weaker performance of UBBA compared to LBBA-bid is due to its load-balancing method being based on the job utilization instead of the worst-case execution time of the jobs, as explained in Section 2.5.

2.6.2. EDF-based Schedulers: Global EDF and EDF-os

Now, in order to compare the behavior of these benefit-aware methods with the state-of-the-art, we schedule the same task set using two very well known, EDF-based algorithms, Global EDF [43] and EDF-os [40]. Global scheduling in case of implicit deadlines is known to be optimal for sporadic task sets, i.e., it can correctly schedule the set (without any time-constraint violations) when there is a correct schedule for that set. However, global scheduling entails higher runtime overheads by allowing any jobs of any tasks to migrate among processors. EDF-os (optimal semi-partitioned EDF) aims to reduce the overhead by partitioning at job-boundary and uses the term optimal meaning that it can correctly schedule a task set, having a guaranteed tardiness bound for each task. In both algorithms, tasks are preemptive and allowed to be tardy, i.e., in case a task misses its deadline, it continues until it completes its execution and will not be discarded.

Under Global EDF scheduling, three jobs, $T_{3,1}$, $T_{1,7}$ and $T_{1,0}$ miss their deadlines and get tardy. Also, the jobs scheduled on P1 cannot finish by the end of LCM (i.e., hyper-period) and the tardiness will propagate through the next LCMs. In the EDF-os schedule, tasks are sorted based on their utilization as $\{T_3$ (0.7), T_2 (0.67), T_1 (0.6)$\}$. T_3 and T_2 are set as fixed tasks and T_1 as a migrating task with 0.3 utilization on the same processor T_3 is scheduled (P2 in Figure 2.3), and the remaining 0.3 on P1. Therefore, P2 is the first processor for T_1 and it gets a higher priority over the fixed tasks on P1 which is not its first processor. Then, T_2, the task with the earliest deadline is a fixed task (non-migrating) and gets lower priority than T_1 which is migrating, having less utilization, but a later deadline.

Consequently, 5 out of 10 jobs of task T_2 (in the first LCM of 30) are tardy, which means 50% tardiness possibility for T_2 with earliest deadline in the example. Also, as shown in Figure 2.3, this tardiness is propagated to the next LCM which can cause even more tardiness in the rest of the schedule.

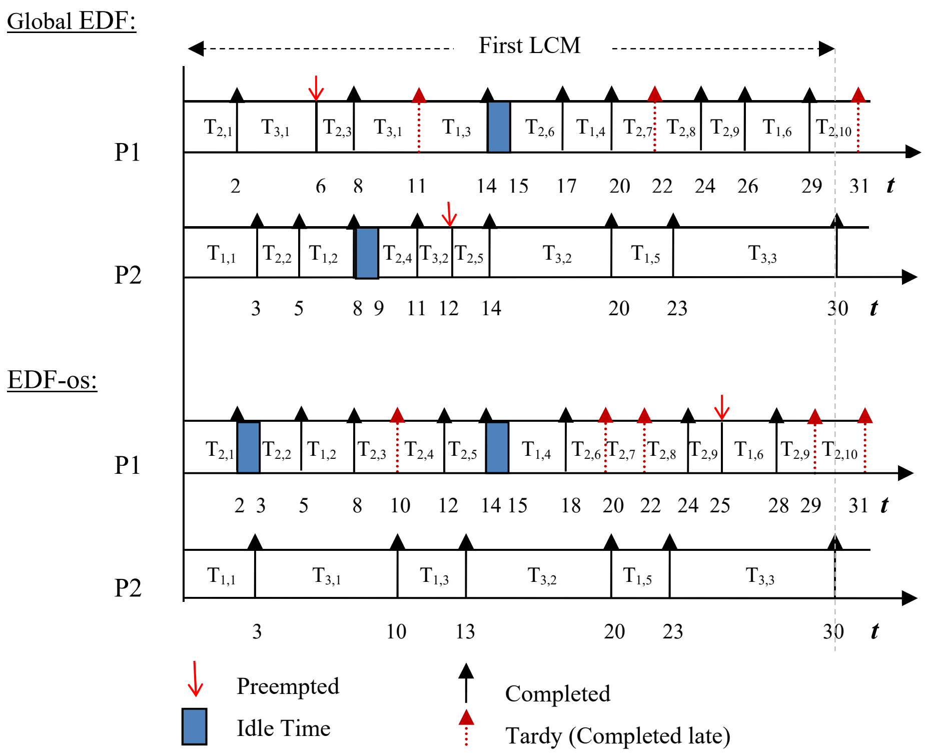

Global EDF:
First LCM
P1
T2,1 T3,1 T2,3 T3,1 T1,3 T2,6 T1,4 T2,7 T2,8 T2,9 T1,6 T2,10
2 6 8 11 14 15 17 20 22 24 26 29 31 t
P2
T1,1 T2,2 T1,2 T2,4 T3,2 T2,5 T3,2 T1,5 T3,3
3 5 8 9 11 12 14 20 23 30 t
EDF-os:
P1
T2,1 T2,2 T1,2 T2,3 T2,4 T2,5 T1,4 T2,6 T2,7 T2,8 T2,9 T1,6 T2,9 T2,10
2 3 5 8 10 12 14 15 18 20 22 24 25 28 29 31 t
P2
T1,1 T3,1 T1,3 T3,2 T1,5 T3,3
3 10 13 20 23 30 t
Preempted
Idle Time
Completed
Tardy (Completed late)

Fig. 2.3: Scheduling diagram for Global EDF and EDF-os

An upper bound for tardiness of fixed jobs under EDF-os scheduling is given in [40]:

"**Theorem 2.** *Suppose that at least one migrating task executes on processor Pp and let τ_i be a fixed task on Pp. If Pp has two migrating tasks (refer to Prop. 3), denote them as τ_h and τ_l, where τ_h has higher priority; otherwise, denote its single migrating task as τ_h, and consider τ_l to be a "null" task with $T_l = 1$, $s_{l,p} = 0$, and $C_l = 0$. Then, τ_i has a maximum tardiness of at most:*

$$\Delta_i = [(s_{h,p})(\Delta_h + 2T_h) + 2C_h + (s_{l,p})(\Delta_l + 2T_l) + 2C_l] / (1 - s_{h,p} - s_{l,p})\text{"}$$

In this example, P1 has only one migrating job and $C_h = 3$, $T_h = 5$, and $\Delta_h = -2$ (lateness of a migrating job can be negative by their definition, i.e., the difference of its completion time and deadline). So the above formula is simplified to:

$$(s_{h,p})(\Delta_h + 2T_h) + 2C_h / 1 - s_{h,p} = (9/30\ (-2 + 10) + 6) / 1 - 9/30$$

$$= 8.4 / 0.7$$

$$= 12$$

This means the upper bound of tardiness for each job of T_2 is 12, while its period is 3. Also, the tardiness bound for the fixed task *i* has no relationship with its deadline in their formula. Also, giving higher priority to the migrating task(s) with smaller utilization than the fixed tasks on the same processor (not the earlier deadline) doesn't follow the EDF scheduling rule. Hence, EDF-os is a partial-EDF scheduling method.

One of the properties of our targeted SRT system model is that there is a specific period of time for each task in which if and only if the task is complete, it is beneficial to the system. In case a job completes after its benefit-aware break point or deadline (arrival (implicit) or completion (non-implicit) of the next job of the same task) whichever is earlier, then not only does a tardy job gain no benefit for the system, but it also wastes the processing time which could be assigned to another job in order to meet its deadline and gain more benefit. Hence, the QoS will be affected (i.e., decreased) by allowing jobs to be tardy, in Global EDF and EDF-os, instead of discarding them after missing their deadlines.

As shown in Figure 2.3, late completion or tardiness of $T_{2,7}$ caused $T_{2,10}$ to get tardy, too. The same scenario repeats for EDF-os when $T_{2,6}$ starts running after its deadline ($t = 18$) and in addition to being tardy, it also prevents $T_{2,7}$, $T_{2,9}$, and $T_{2,10}$ from meeting their deadlines. Nevertheless, if $T_{2,6}$ was discarded at its deadline, three latter jobs we mentioned could have finished on time and also $T_{2,9}$ would not have been preempted. Reducing the missed deadline ratio and number of preemptions can enhance a system's QoS.

Therefore, in order to make a better judgment in the empirical comparison of our benefit-aware methods with Global EDF and EDF-os, we implemented their scheduling methods but with the same firm benefit-aware deadlines considered in our methods, to see which one could gain more benefit, with less preemptions and lower missed deadlines ratio. The details of our simulation experiments are provided in the next section.

2.7. Experimental Evaluation

Through extensive experiments on synthetic periodic task sets, we conducted an comparative performance evaluation for the three proposed algorithms, LBBA-bid, BBA-bnc, UBBA, and two other state-of-the-art algorithms, Global EDF with global scheduling approach [43] and EDF-os which is a semi-partitioned scheduling [40]. We compare the schedulability (job completion rate), job flow time, gained benefit, and the number of preemptions in the proposed algorithms with Global EDF and EFD-os, which are known as optimal methods for scheduling periodic tasks, to show how close our benefit-aware scheduling methods are to the optimal solution, in term of schedulability, while increasing the total benefit gained, and reducing cost by decreasing flow time and preemptions.

2.7.1. Performance Metrics

In this work, we consider the following measurements to evaluate and compare the performance of the three proposed algorithms, plus Global EDF and EDF-os.

For each task set in its LCM, we measure:

- ***Average Benefit per job*** = $\frac{Total\ Gained\ Benefit\ (B)}{Number\ of\ jobs\ (N)}$

Note: We consider this benefit accrual measurement for all the algorithms in our experiments, even Global EDF and EDF-os. Our objective for measuring gained benefit in Global EDF and EDF-os algorithms, claimed to have optimal schedulability, is to evaluate their performances for the systems in which tasks have different benefit density functions.

- ***Avg. Preemptions per job*** = $\frac{Total\ number\ of\ preemptions}{N}$

- ***Schedulability*** = $\frac{Number\ of\ jobs\ completed\ by\ deadline}{N}$

- ***Avg. Flow time Stretch*** = $\frac{Sum\ of\ (stretch\ ratio)\ for\ all\ jobs}{N}$

(*Stretch ratio* of job $T_{i,j}$ = $\frac{Flow\ time\ (fj)}{Execution\ time\ (wj)}$)

2.7.1. Experimental Setting

We implemented the algorithms using Netbeans 8.1, on Intel core i7- 6700HQ CPU at 2.6 GHz speed, 64 bit OS, 16 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. Three different benefit density functions, $\frac{1}{x}, \frac{1}{2x}, \frac{1}{x^2}$ were assigned to the tasks, and the experiments were repeated for systems with 75% and nearly 100% utilizations. Task sets were generated with a 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]

For simulating systems with nearly full utilization, we generated the tasks until the total utilization was in the range of [90%, 100%]. We ran hundreds of trials for each multiprocessor setting and calculated the average amount of recorded results for the metrics.

2.7.3. Results and Discussion

The results of our extensive experiments are shown in Figures 2.4 through Figure 2.9. We discuss the results of our comparisons based on our performance metrics as follows:

a) Average Preemptions per Job

In all of our algorithms, the average numbers of preemptions were very close and the results were shown as overlapping lines in the graphs for 75% utilizations and almost the same as Global EDF. However, for near full utilization, UBBA showed a slightly better performance. Our methods improved (i.e., decreased) the results of EDF-os in near full utilization systems as listed below (See Figure 2.4 (a) and (b). P stands for processors):

• UBBA:	From 54% (2P) to 85% (8P)
• LBBA-bid:	From 49% (2P) to 77% (8P)
• UBBA-bnc:	From 44% (2P) to 72% (8P)

b) Flow Time Stretch Ratio

The flow time stretch ratio shows how much longer than its WCET, in average, each job takes to complete. For example, 1.11 means that flow time is 11% longer than WCET. As can be seen in the graphs, LBBA-bid had the best performance in reducing flow time in systems with near full utilization. It showed more than 50% improvement (6 and 8P) to 74% (2P) compared to Global EDF, and from 48% (6 and 8P) to 64% (2P) improvement compared to EDF-os, both on near full utilization systems (See Figure 2.5 (a) and (b)).

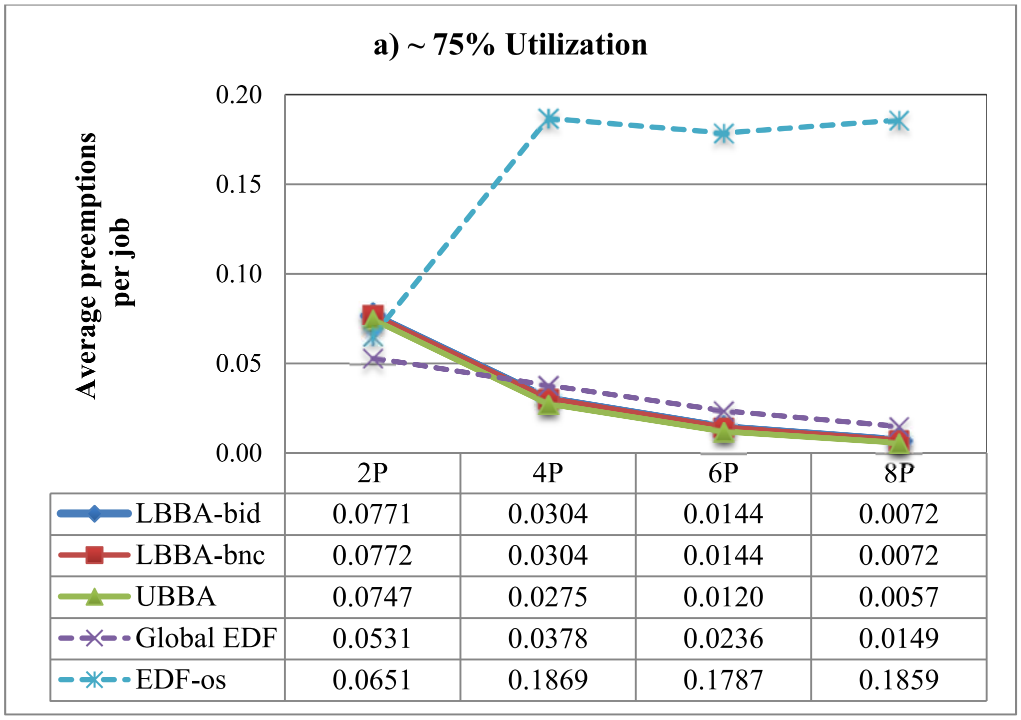
c) Schedulability and Missed Deadline Ratio

LBBA-bid for implicit deadlines, scheduled 99.7% (2P) to 99.9% (8P) in 75% utilization, and 90% (2P) to 95% (8P) in near full utilization. Schedulability of LBBA-bnc (for tasks with non-implicit deadlines) was 100% for the systems with 75% utilization, and from 99% (2P) to 99.64% (8P) for near full utilization (Figure 2.6 (a) and (b)). These results show that our benefit-based algorithms outperform the state-of-the-art, e.g., EDF-os, from 11% to 20%.

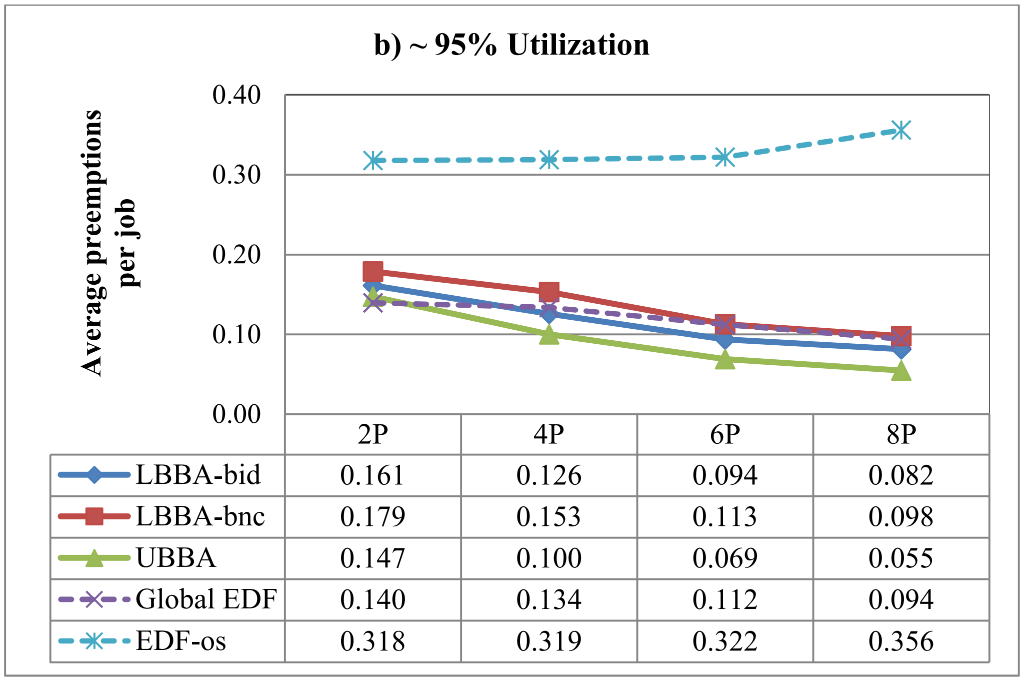
The missed deadline ratio (Figures 2.8 and 2.9) of less than 1% in LBBA-bnc can be negligible, having the maximum benefit per job gained by LBBA-bnc among all the tested algorithms and considering the fact that these results are for the worst-case execution time of the tasks, and in real cases tasks may take shorter time to complete.

d) Average Benefit per Job

In our simulation experiments, as the utilization increased, our proposed algorithms outperformed the others. The benefit gained by LBBA-bid and LBBA-bnc were up to 12.5% more than the others for 2P, and 20% more for 8P in near full utilization systems (Figure 2.7).

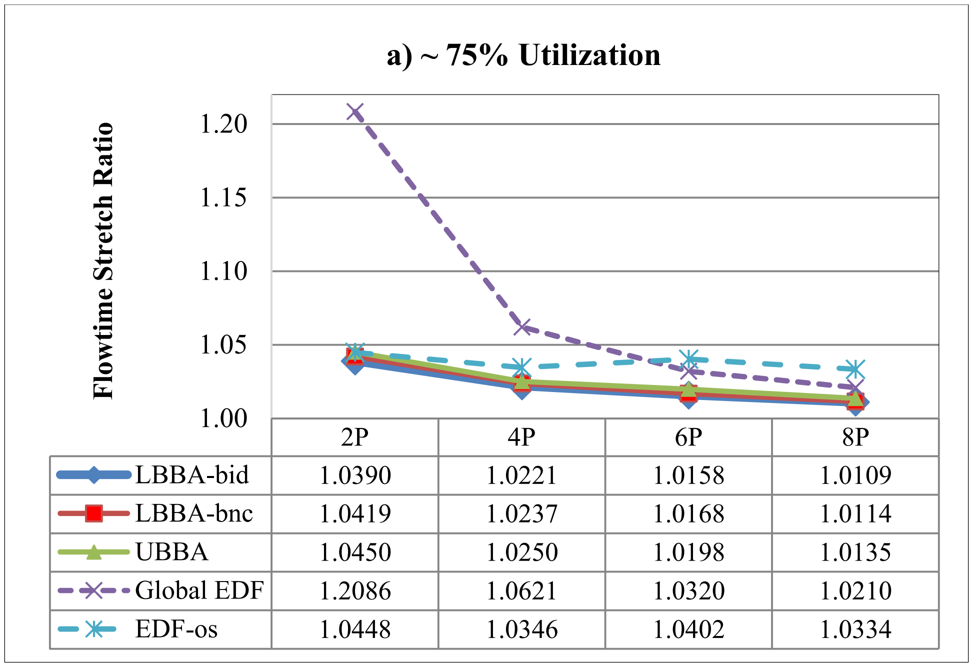


	2P	4P	6P	8P
◆ LBBA-bid	0.0771	0.0304	0.0144	0.0072
■ LBBA-bnc	0.0772	0.0304	0.0144	0.0072
▲ UBBA	0.0747	0.0275	0.0120	0.0057
× Global EDF	0.0531	0.0378	0.0236	0.0149
✳ EDF-os	0.0651	0.1869	0.1787	0.1859

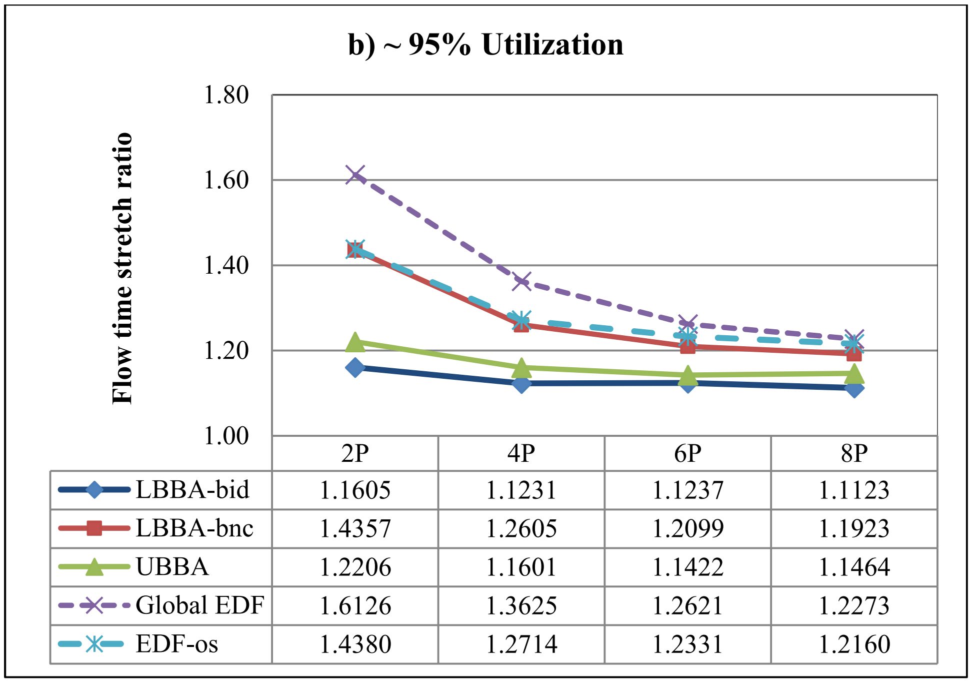


	2P	4P	6P	8P
◆ LBBA-bid	0.161	0.126	0.094	0.082
■ LBBA-bnc	0.179	0.153	0.113	0.098
▲ UBBA	0.147	0.100	0.069	0.055
× Global EDF	0.140	0.134	0.112	0.094
✳ EDF-os	0.318	0.319	0.322	0.356

Fig. 2.4: Average preemptions per job versus number of processors in the systems



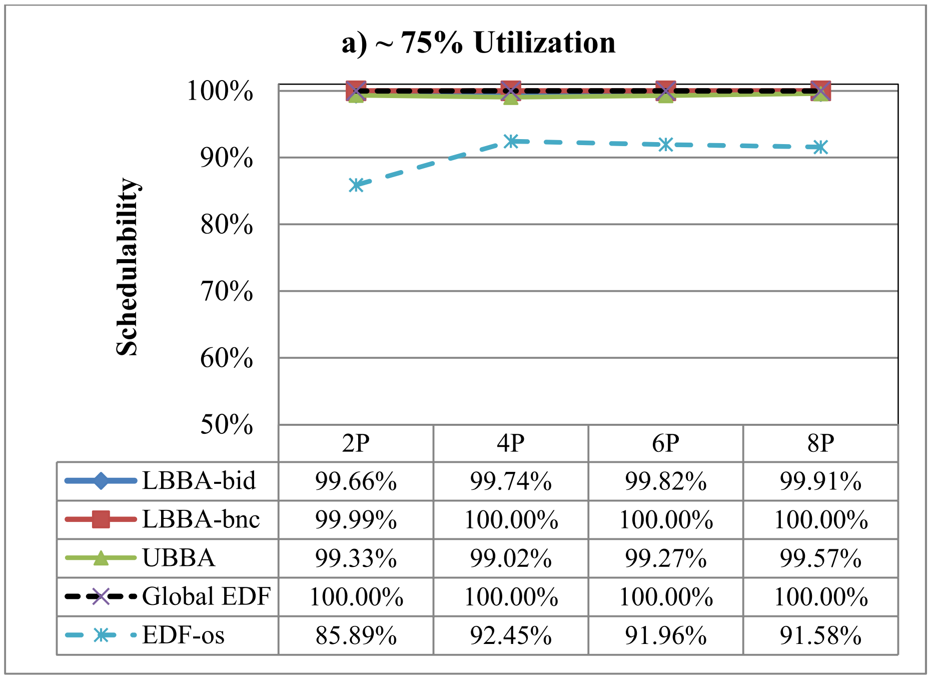
	2P	4P	6P	8P
LBBA-bid	1.0390	1.0221	1.0158	1.0109
LBBA-bnc	1.0419	1.0237	1.0168	1.0114
UBBA	1.0450	1.0250	1.0198	1.0135
Global EDF	1.2086	1.0621	1.0320	1.0210
EDF-os	1.0448	1.0346	1.0402	1.0334



	2P	4P	6P	8P
LBBA-bid	1.1605	1.1231	1.1237	1.1123
LBBA-bnc	1.4357	1.2605	1.2099	1.1923
UBBA	1.2206	1.1601	1.1422	1.1464
Global EDF	1.6126	1.3625	1.2621	1.2273
EDF-os	1.4380	1.2714	1.2331	1.2160

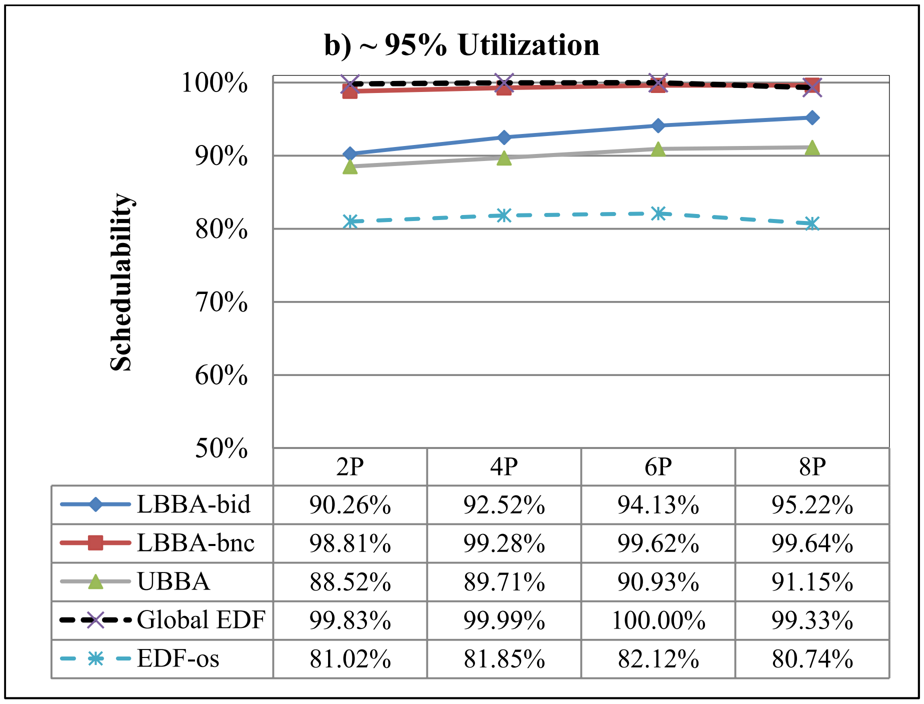
Fig. 2.5: Average flow time stretch ratio per job vs. number of processors in the systems

a) ~ 75% Utilization



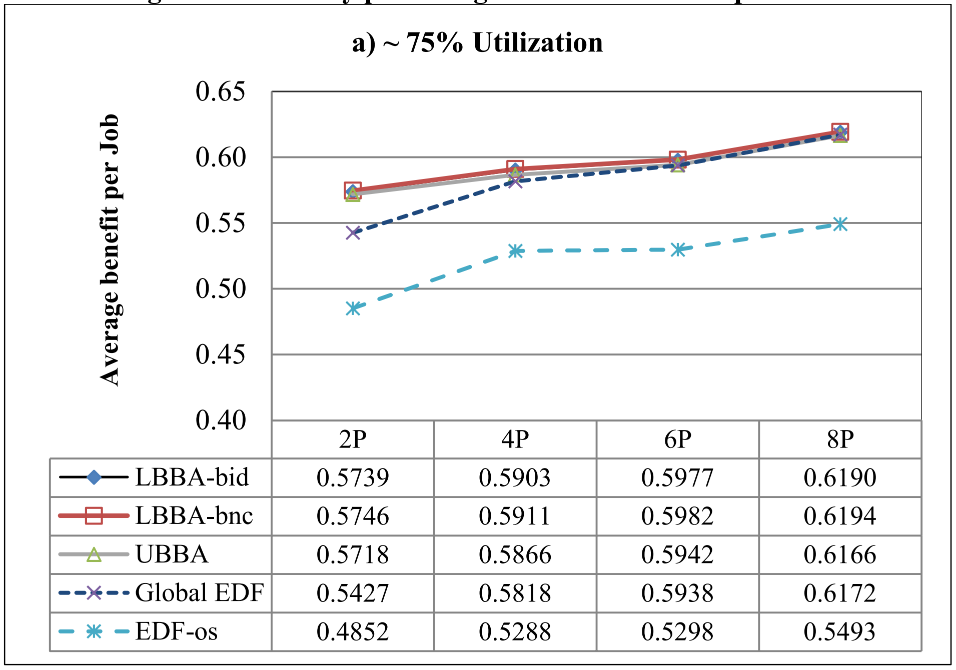
	2P	4P	6P	8P
LBBA-bid	99.66%	99.74%	99.82%	99.91%
LBBA-bnc	99.99%	100.00%	100.00%	100.00%
UBBA	99.33%	99.02%	99.27%	99.57%
Global EDF	100.00%	100.00%	100.00%	100.00%
EDF-os	85.89%	92.45%	91.96%	91.58%

b) ~ 95% Utilization

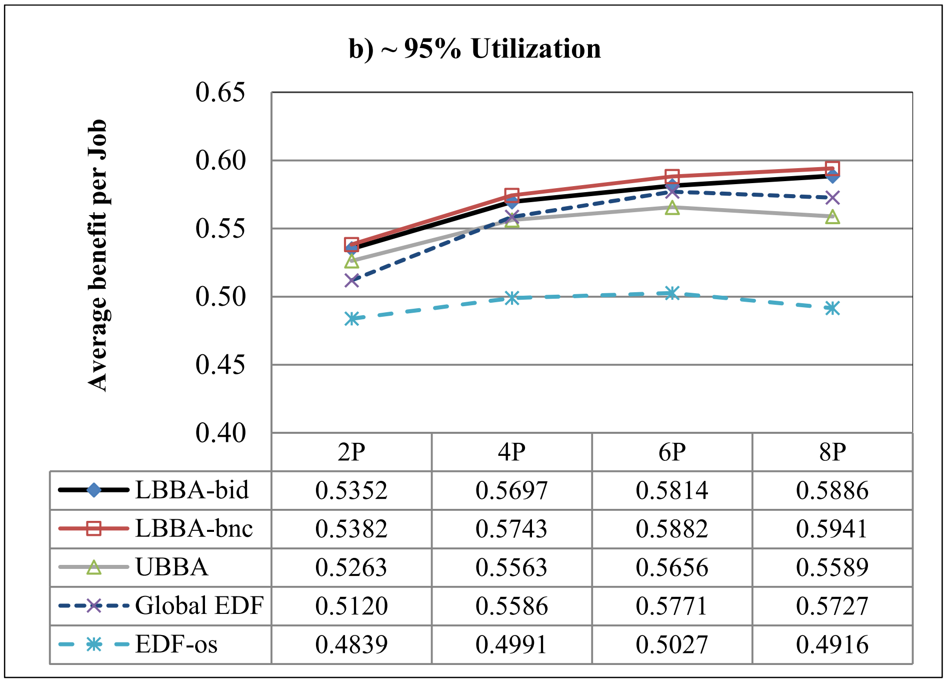


	2P	4P	6P	8P
LBBA-bid	90.26%	92.52%	94.13%	95.22%
LBBA-bnc	98.81%	99.28%	99.62%	99.64%
UBBA	88.52%	89.71%	90.93%	91.15%
Global EDF	99.83%	99.99%	100.00%	99.33%
EDF-os	81.02%	81.85%	82.12%	80.74%

Fig. 2.6: Average schedulability percentage versus number of processors in the system



	2P	4P	6P	8P
—◆— LBBA-bid	0.5739	0.5903	0.5977	0.6190
—□— LBBA-bnc	0.5746	0.5911	0.5982	0.6194
—△— UBBA	0.5718	0.5866	0.5942	0.6166
--×-- Global EDF	0.5427	0.5818	0.5938	0.6172
– ✳ – EDF-os	0.4852	0.5288	0.5298	0.5493



	2P	4P	6P	8P
—◆— LBBA-bid	0.5352	0.5697	0.5814	0.5886
—□— LBBA-bnc	0.5382	0.5743	0.5882	0.5941
—△— UBBA	0.5263	0.5563	0.5656	0.5589
--×-- Global EDF	0.5120	0.5586	0.5771	0.5727
– ✳ – EDF-os	0.4839	0.4991	0.5027	0.4916

Fig. 2.7: Average benefit per job versus number of processors in the system

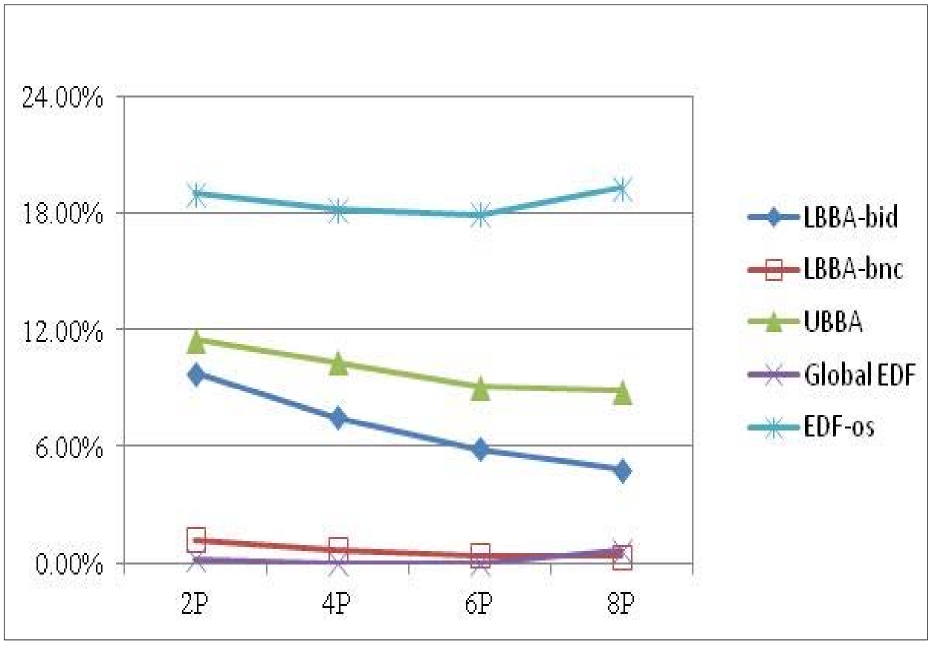


Fig. 2.8: Missed deadline ratio versus number of processors (~ 95% Utilization)

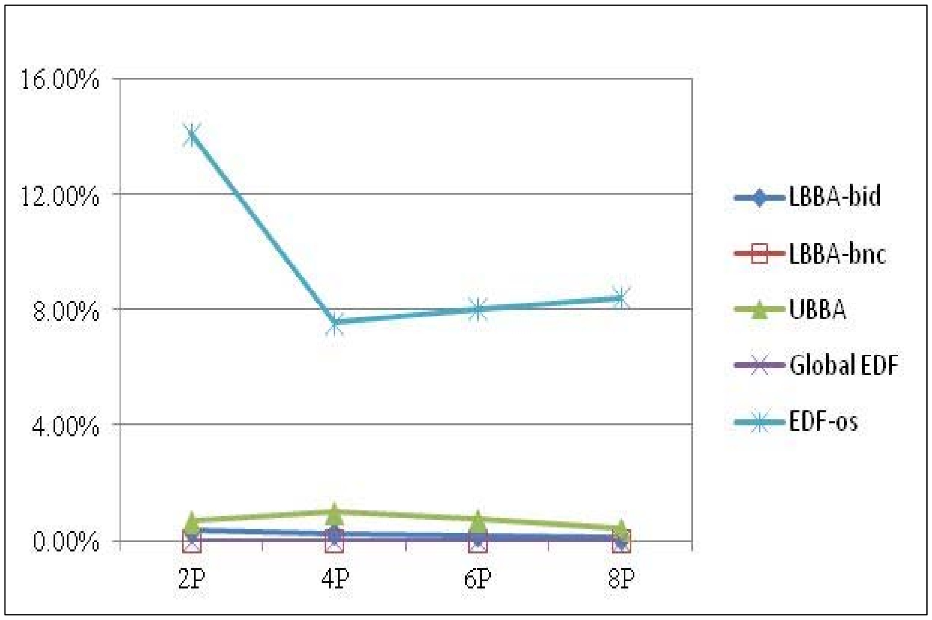


Fig. 2.9: Missed deadline ratio versus number of processors (~ 75% Utilization)

2.8. Summary

In this Chapter, I proposed a new semi-partitioning approach to schedule soft real-time periodic task sets on identical multiprocessor systems to enhance their QoS. 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.

This hybrid method for scheduling periodic tasks is a combination of benefit model and cost model which increases the total benefit while balancing the workload among the processors for reducing flow time and deadline misses. In this dissertation, the upper bounds on the delays and preemptions in accordance to the task utilizations, and schedulability conditions of periodic tasks were provided.

In addition, experimental performance analysis was conducted for the proposed algorithms, LBBA-bid for periodic tasks with implicit deadlines, LBBA-bnc with non-implicit deadlines, and UBBA with utilization-balancing, compared to the-state-of-the-art, Global EDF and EDF-os, in terms of total gained benefit, job completion rate or schedulability, number of preemptions per job and flow time stretch ratio.

In these simulation experiments, LBBA-bid showed the best performance among the algorithms for implicit deadlines, resulting in the highest amount of gained benefit. LBBA-bnc for tasks with non-implicit deadlines, allows parallel processing of two consecutive jobs of the same task until they can complete in the same order they are released, i.e., each job of a task has to complete before the next job of that task. Otherwise, it will be discarded. This relaxation of deadline in LBBA-bnc, provided a near optimal schedulability in the conducted experiments, without the runtime overhead of task migrations (during execution of any task instance) allowed in Global EDF, while having almost the same benefit as gained by LBBA-bid.

As the number of processors and utilization of the system increases, the advantage of using the proposed models for reducing number of preemption, decreasing flow times, increasing gained benefit and schedulability is more substantial, and overall, they can enhance QoS in systems with SRT periodic tasks.

3. Conclusion and Perspectives

3.1. List of Contributions

In our research, we were interested in online benefit-aware preemptive multiprocessor scheduling of soft real-time tasks. Benefit-awareness in scheduling is very essential for the Quality of service (QoS) in soft real-time applications and embedded system with tasks that are not equally critical or beneficial to the system. These applications may have aperiodic, periodic or mixed task sets. Some examples of such applications are medical monitoring systems and video surveillance which have both periodic and aperiodic tasks. They periodically receive data, analyze and record it. However, if they receive and process an abnormal data, they must send alert which is an aperiodic task.

In this work, we first proposed a novel solution for aperiodic task sets (called LBBA). The proposed scheduler is a hybrid technique, combining benefit and cost models, which improves the quality of service in the systems, by gaining more benefit at lower cost. We introduced an online choice of approximation algorithms for partitioning lower priority tasks among the processors while the higher priority tasks get scheduled as soon as they are released. LBBA is superior to other existing methods (such as BBA [29]) in principle, since:

- **LBBA is a novel hybrid model** of soft real-time multiprocessor scheduling. In contrast to BBA, which only follows a benefit model, LBBA is a combination of benefit and cost models. That is, it aims to **minimize makespan** in order to achieve the *maximum benefit at the lowest cost*.

- **No synchronization is needed** for fetching the jobs from a shared pool. That is because a separate pool is assigned to each processor in contrast to the other method (BBA) where all processors use a shared pool.
- **LBBA facilitates load-balanced partitioning** of waiting jobs, while this case is not considered in BBA.
- **LBBA optimizes the CPU usage and minimizes the total idle time** of the processors by balancing the workload among them.
- **LBBA improves Quality-of-Service (QoS) by reducing missed deadline ratio:** LBBA reduces the possibility of starvation for low priority jobs, comparing to BBA. It also has a **Minimal Response time,** including both scheduling and execution time, for a job set (up to 300 times faster response time than BBA in our experiments shown in 1.4.1.2.).
- LBBA is computationally less expensive than BBA, as we prove in sub-section 1.3.2.

In the second part of our work, we proposed benefit-aware multiprocessor scheduling methods for soft real-time periodic tasks with implicit and non-implicit deadlines, called LBBA-bid and LBBA-bnc, respectively. We empirically compared our solutions with the state-of-the-art (Global EDF and EDF-os) and the results of our simulation experiments showed superiority of our methods in the sense of gaining more benefits per job, less preemptions, shorter flow times and 90 to 95 percent schedulability in systems with near full utilization (for implicit deadlines) and above 99% schedulability for tasks with non-implicit deadlines.

3.2. Future Work and Perspectives

For further research, our suggestions are as follows:

a) General Framework for Benefit-Aware Multiprocessor Scheduling of SRT Tasks

Based on the conclusion, I propose a general framework for benefit-aware multiprocessor scheduling in a SRT system of aperiodic, periodic or mixed tasks to be analyzed, implemented and evaluated in future work. I define the framework as follows:

In order to support QoS enhancement for a wider domain of soft real-time applications and embedded systems, I propose a multi-mode framework to be adopted by multiprocessor systems with any combination of aperiodic and/or periodic, SRT tasks (i.e., each task can be aperiodic or one instance, periodic with implicit deadline or periodic with non-implicit deadline). Tasks will dynamically select one of the three scheduling modes as soon as they are released (see Figure 3.1).

This framework applies the LBBA model for aperiodic tasks, LBBA-bid for periodic tasks which have implicit deadlines, and LBBA-bnc for scheduling periodic tasks with non-implicit deadlines. LBBA-bid and LBBA-bnc are considered for scheduling periodic tasks in this general framework, since they showed the best overall performance in our extensive experimental evaluations compared to the other methods. Figure 3.1 is a schema of the framework. Every task arrives with an index showing its attribute (AP for aperiodic, periodic with implicit deadline or PN for periodic with non-implicit deadline).
CTA will be a built-in function in the framework which checks the value of the attribute index and based on the value directs the task to the appropriate scheduler.

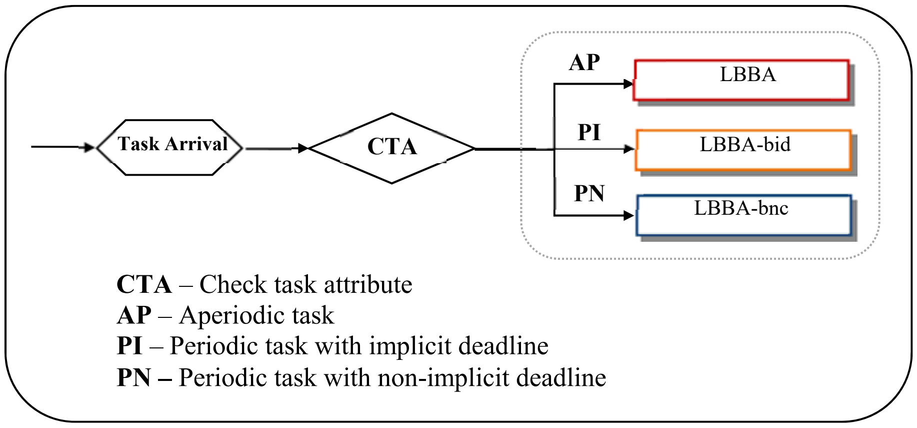

Task Arrival
CTA
AP
PI
PN
LBBA
LBBA-bid
LBBA-bnc
CTA – Check task attribute
AP – Aperiodic task
PI – Periodic task with implicit deadline
PN – Periodic task with non-implicit deadline

Fig. 3.1: General framework for SRT benefit-aware multiprocessor scheduling

In Figure 3.1, the framework is shown as a package of three separate schedulers, LBBA, LBBA-bid, and LBBA-bnc. However, many functions such as priority assignment, benefit-aware scheduling, load-balanced partitioning and benefit calculation, are the same in all three methods. Hence, for code optimization, all three schedulers can share the functions that are common among them, and only the functions responsible for setting the deadlines (according to the task attributes) and checking for missed deadlines will be implemented separately.

b) SRT Applications and Embedded Systems

These proposed methods can be applied to actual SRT applications and embedded systems, such as video streaming or RT medical monitoring systems, with periodic and/or mixed tasks for more evaluations.

c) Task Dependencies

Another worthwhile extension of this research is studying models with inter-task dependencies such as precedence constraints. Such tasks can be shown in a Direct Acyclic Graph (DAG).

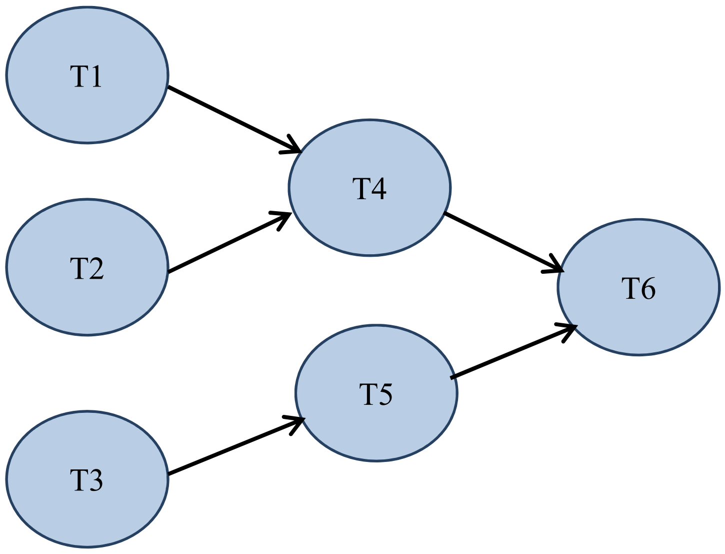


Fig. 3.2: An example of a DAG task set

References

[1] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, Edited by D. S. Hochbaum, "Approximation Algorithms for NP-Hard Problems," Chapter 2: "Approximation Algorithms for Bin Packing: A Survey," *page 67, 1997.*

[2] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal on Applied Mathematics, 17:263-269, 1969.*

[3] L. Welch and S. Brandt, "Toward a Realization of the Value of Benefit in Real-Time Systems," *Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2001), San Francisco, California, April 23-24, 2001.*

[4] R. Rajkumar, C. Lee, J. P. Lehozcky, and D.P. Siewiorek, "A Resource Allocation Model for QoS Management," *Proc. 18th IEEE Real-Time Systems Symposium, pp. 298-307, December 1997.*

[5] E. Chang and A. Zakhor, "Scalable Video Coding Using 3-D Subband Velocity Coding and Multi-Rate Quantization," *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing, pp. 574-577, July 1993.*

[6] E. Chang and A. Zakhor, "Scalable Video Data Placement on Parallel Disk Data Arrays," *Proc. ISIT/SPIE Symp. Electronic Imaging Science and Technology, pp. 208-223, February 1994.*

[7] W. Feng and J. W. S. Liu, "An Extended Imprecise Computation Model for Time-Constrained Speech Processing and Generation," *Proc. IEEE Workshop Real-Time Applications, pp. 76-80, May 1993.*

[8] C. J. Turner and L. L. Peterson, "Image Transfer: An End-to-End Design," *Proc. SIGCOMM Symp. Comm. Architectures and Protocols, pp. 258-268, August 1992.*

[9] M. Boddy and T. Dean, “Solving Time-Dependent Planning Problems,” *Proc. 11th Int'l Joint Conf. Artificial Intelligence (IJCAI-89), pp. 979-984, August 1989.*

[10] B. Hayes-Roth, “Architectural Foundations for Real-Time Performance in Intelligent Agents,” *Journal of Real-Time Systems, vol. 2, no. 1, pp. 99-125, 1990.*

[11] S. Zilberstein and S. J. Russell, “Anytime Sensing, Planning and Action: A Practical Model for Robot Control,” *Proc. 13th Int'l Joint Confs. Artificial Intelligence, pp. 1402-1407, 1993.*

[12] E. J. Horvitz, “Reasoning under Varying and Uncertain Resource Constraints,” *Proc. Seventh Nat'l Conf. Artificial Intelligence (AAAI-88), pp. 111-116, August 1988.*

[13] J. Grass and S. Zilberstein, “A Value-Driven System for Autonomous Information Gathering,” *Journal of Intelligent Information Systems, vol. 14, pp. 5-27, March 2000.*

[14] R. E. Korf, “Real-Time Heuristic Search,” *Artificial Intelligence, vol. 42, no. 2, pp. 189-212, 1990.*

[15] S. V. Vrbsky and J. W. S. Liu, “APPROXIMATE: A Query Processor that Produces Monotonically Improving Approximate Answers,” *IEEE Trans. Knowledge and Data Eng., vol. 5, no. 6, pp. 1056-1068, December 1993.*

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

[17] S. Irani and Anna R. Karlin, edited by D.S. Hochbaum, “Approximation Algorithms for NP-Hard Problems,” Chapter 13: “Online Computation,” *page 552, 1997.*

[18] C. Y. Yang, J. Chen, and T. W. Kuo, “An Approximation Algorithm for Energy-Efficient Scheduling on A Chip Multiprocessor,” *ACM/IEEE Design, Automation, and Test in Europe (DATE), Munich, Germany, March 2005.*

[19] J. J. Chen, T. W. Kuo, and C. Y. Yang, "Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems," *the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Bellevue, WA, USA, April 3-6, 2007.*

[20] B. Sanati and A. M. K. Cheng, "Maximizing Job Benefits on Multiprocessor Systems Using a Greedy Algorithm," *WiP Session of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2008.*

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

[22] C. F. Kuo, T. W. Yang, and T. W. Kuo, "Dynamic Load Balancing for Multiple Processors," *IEEE 12th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2006), Sydney, Australia, August 16-18, 2006.*

[23] E. Piel, P. Marquet, J. Soula, and J. L. Dekeyser, "Load-Balancing for a Real-Time System Based on Asymmetric Multi-Processing," *The 16th Euromicro Conference on RealTime Systems, WiP, April 2004.*

[24] S. A. Brandt and G. J. Nutt, "Flexible Soft Real-Time Processing in Middleware," *Journal of Real- Time Systems, Kluwer, 2001.*

[25] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the Real-Time Systems Symposium*, 112-122, *IEEE CS Press, 1985.*

[26] L. R. Welch, B. Ravindran, B. Shirazi, and C. Bruggeman, "Specification and Analysis of Dynamic, Distributed Real-Time Systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium, 72-81, IEEE Computer Society Press, 1998.*

[27] D. Andrews, L. R. Welch, and S. Brandt, "A Framework for Using Benefit Functions In Complex Real Time Systems", *Journal of Parallel and Distributed Computing Practices*, *Volume 5, No. 1, March 2002.*

[28] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions," in *Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1998.*

[29] B. Awerbuch, Y. Azar, and O. Regev, "Maximizing Job Benefits On-Line", *Proceedings of the third International Workshop, APPROX, Germany, September 2000.*

[30] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Optimal Reward-Based Scheduling For Periodic Real-Time Tasks," *IEEE Transactions On Computers, vol. 50, no. 2, February 2001.*

[31] J.K. Dey, J. Kurose, and D. Towsley, "On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks," *IEEE Trans. Computers, vol. 45, no. 7, pp. 802-813, July 1996.*

[32] B. Sanati and 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, 2016.*

[33] A. Elnably, K. Du, and 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.*

[34] 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.*

[35] H. Alhussian, N. Zakaria, and F. A. Hussin, "An Efficient Real-Time Multiprocessor Scheduling Algorithm," *Journal of Convergence Information Technology*, *January 2014.*

[36] 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.*

[37] I-H. Hou, and P.R. Kumar, ”Scheduling Periodic Real-Time Tasks with Heterogeneous Reward Requirements,” in *Proc. of the 32nd IEEE Real-Time Systems Symposium*, *2011.*

[38] Ming Zu and Albert M. K. Cheng, “ Real-Time Scheduling of Hierarchical Reward-Based Tasks," in *Proc. of IEEE-CS Real-Time Technology and Applications Symposium, May 2003.*

[39] J. J. Chen, T. W. Kuo, and C. L. Yang, “Profit-Driven Uniprocessor Scheduling with Energy and Timing Constraints,” in *Proc. of the ACM Symposium on Applied Computing, Nicosia, Cyprus, pp. 834 – 840, 2004.*

[40] J. H. Anderson, J. P. Erickson, U. C. Devi, and 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.*

[41] J. J. Chen, and S. Chakraborty, “Partitioned Packing and Scheduling for Sporadic Real-Time Tasks in Identical Multiprocessor Systems,” in *Proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, *pp. 24-33, 2012*.

[42] A. M. K. Cheng, and F. Shang, "Priority-Driven Coding and Transmission of Progressive JPEG Images for Real-Time Applications," *Journal of VLSI Signal Processing-Systems for Signal, Image and Video Technology, vol. 47, pp. 169-182, 2007.*

[43] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, “Global EDF Scheduling for Parallel Real-Time Tasks,” *Real-Time Systems, vol. 51, no. 4, pp. 395-439, 2015.*

[44] B. Sanati, A.M.K. Cheng, and N. Troutman, “Online Benefit-Aware Semi-Partitioned Scheduling of Periodic Soft Real-Time Tasks for QoS Enhancement”, Technical Report no. UH-CS-16-02, University of Houston, Department of Computer Science, *2016*.