Technical Report Number 2007-535

Number of Processors for Scheduling a Set of Real-Time Tasks: Upper and Lower Bounds*

Arezou Mohammadi and Selim G. Akl
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6
E-mail:{arezoum, akl}@cs.queensu.ca

June 13, 2007

Abstract

In this report, we study the problem of scheduling a set of n periodic preemptive independent hard real-time tasks on the minimum number of processors. We assume that the partitioning strategy is used to allocate the tasks to the processors and the EDF method is used to schedule the tasks on each processor. It is known that this problem is NP-hard; thus, it is unlikely to find a polynomial time algorithm to schedule the tasks on the minimum number of processors.

In this work, we derive a lower and an upper bound for the number of processors required to satisfy the constraints of our problem. We also compare a number of heuristic algorithms with each other and with the bounds derived in this report. Numerical results demonstrate that our lower bound is very tight and it is very close to the optimal solution.

Keywords: Periodic Hard Real-Time Tasks, Real-Time Scheduling, Minimum Number of Processors, Lower Bound, Upper Bound, Partitioning Strategy, EDF-Schedulability Test.

---

*This work was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada.

1 Introduction

The purpose of a real-time system is to produce a response within a specified time-frame. In other words, for a real-time system not only the logical correctness of the system should be satisfied, but also it is required to fulfill the temporal constraints of the system. Typically, a real-time system consists of a controlling system (e.g. a computer) and a controlled system (e.g. an environment). The controlling system interacts with its environment. On a real-time computer which controls a device or process, sensors will provide readings at periodic intervals and the computer must respond by sending signals to actuators. There may be unexpected or irregular events and these must also receive a response. In all cases, there will be a time bound within which the response should be delivered. A real-time application is normally composed of multiple tasks with different levels of criticality. Failure to meet the timing constraint for a response can have different consequences; although missing deadlines is not desirable in a real-time system, some real-time tasks could miss some deadlines and the system will still work correctly while certain penalties will have to be paid for the deadlines missed. On the other hand, some real-time tasks cannot miss any deadlines, otherwise, undesirable or fatal results will be produced in the system [13, 24, 17, 23]. The latter class, called *hard real-time tasks*, is the type of real-time tasks which we consider in this report.

The ability of a computer to meet the timing constraints depends on its capacity to perform the necessary computations. If a number of events occur close together, the computer will need to schedule the computations so that each response is provided within the required time bounds. It may be that, even so, the system is unable to meet all possible unexpected demands. In order to prevent failure of the system, one should increase the number of the processors. Having a sufficient number of processors, one is able to schedule the tasks without missing any deadlines.

Multiprocessor scheduling algorithms are categorized into either Partitioning or Global strategy. In this report, we focus our attention on algorithms that use a partitioning strategy. Partitioning strategies reduce a multiprocessor scheduling problem to a set of uniprocessor ones, thereby allowing well-known uniprocessor scheduling algorithms to be applied to each processor. Using the partitioning approach, we need to consider both an allocation algorithm to determine the processor that should be assigned to each task and a scheduling algorithm to schedule the tasks assigned on each processor. One of the main concerns in designing a partitioning strategy is finding an algorithm to allocate the tasks to the minimum number of processors that are required.

Many methods have been devised to make a good combination of the pair of (allocation, scheduling) algorithms. The most popular scheduling algorithms are the Rate

Monotonic (RM) and the Earliest Deadline First (EDF) algorithms, since both of them are optimal scheduling algorithms on uni-processor systems [16]. By optimality, we mean that if an optimal algorithm cannot schedule a task set on a processor such that deadlines are met, there is no other scheduling algorithm that can do so.

Some researchers have focused on the problem when RM scheduling is used on each processor (see, for example, [19, 20]). However, our concern is solving the problem by employing a partitioning method, when the EDF scheduling algorithm is used on each processor and the allocation algorithm fits into the following frame. In this report, we generally talk about any allocation algorithm which picks up tasks one by one and assigns each task to one of the existing processors and if there is not enough room on them for the new task, then we add a new processor. It is proved in [1, 10, 15] that the problem is reducible to the Bin Packing problem (BPP), and consequently the problem is NP-hard. Therefore, the most efficient known algorithms use heuristics, which may not be the optimal solution, to accomplish very good results in most cases. The Best-Fit (BF), First-Fit (FF), Best-Fit Decreasing (BFD), First-Fit Decreasing (FFD), Modified First-Fit Decreasing (MFFD), Next-Fit (NF), Annealing Genetic (AG) algorithms are a number of heuristic algorithms derived so far for the problem [14, 5, 6, 21].

In [3], it is proved that the worst-case achievable utilization (see Section 2) on M processors for all of the above-mentioned heuristics (and also for an optimal partitioning algorithm) is only $(M + 1)/2$, even when an optimal uniprocessor scheduling algorithm such as EDF is used. In other words, there exist task systems with utilization slightly greater than $(M+1)/2$ that cannot be correctly scheduled by any partitioning approach.

It is proved in [18] that the problem of finding the minimum number of processors with l priority levels to schedule a set of tasks is NP-hard.

Let an *implementation* consist of a hardware platform and the scheduler under which the program is executed. An implementation is said to be *feasible* if every execution of the program meets all its deadlines. Based on the properties of the real-time system, the parameters of the system, and the algorithm applied for scheduling, we may determine sufficient conditions for feasibility of the scheduling algorithm. In what follows, by *schedulability test*, we mean checking whether the sufficient conditions of a given scheduling algorithm hold for a set of real-time tasks. For instance, for a set of periodic hard real-time tasks, there exist simple schedulability tests corresponding to the case where the relative deadlines are all equal to the periods. In such a case, if all tasks are periodic and have relative deadlines equal to their periods, they can be feasibly scheduled by the EDF algorithm *if and only if* the utilization factor of the set of the tasks is smaller than or equal to 1. In this report, we use EDF scheduling on each processor. Therefore, we should look for an appropriate allocation algorithm to assign the tasks to the minimum

number of processors while the EDF-schedulability condition is satisfied on each processor. The problem is reducible to the BPP, and therefore, we take advantage of the existing heuristic algorithms for BPP. We derive upper and lower bounds for the number of processors required by any allocation algorithm with the EDF schedulability test for each processor.

We compare our bounds with a number of heuristic algorithms using simulations. As expected, the number of processors as determined by the heuristic algorithms is bounded by our upper and lower bounds.

The upper bound derived here is for the number of processors achieved by any allocation algorithm in our framework and with EDF-schedulability test on each processor. Therefore, the number of processors required by the allocation algorithms should not exceed the upper bound derived in the paper.

The lower bound that is obtained in this report is very tight and its results are very close to the results of the FFDU and BFDU algorithms discussed in Section 6.1. The minimum number of the processors required for scheduling is equal to or larger than the proposed lower bound for the number of processors and smaller than the best heuristic algorithm. Since we look for the algorithm with the minimum number of processors, the best heuristic algorithm amongst the above heuristic algorithms is the one with the least guaranteed performance. Simulation results lead us to the fact that the difference between the number of processors achieved by the FFDU and BFDU algorithm and the results of an optimal algorithm is negligible.

The remainder of this paper is organized as follows. We introduce terminology in Section 2. Then, in Sections 3 and 4, the EDF scheduling algorithm and the Bin Packing problem and a number of its approximation algorithms are discussed, respectively. In Section 5, we formally define the problem under study. In Section 6, we discuss a number of heuristic algorithms and derive the upper and lower bounds for the number of processors. In Section 7, we present simulation results and compare the performance of the heuristic algorithms with the upper and lower bounds derived in this report. Section 8 contains the conclusions.

2 Terminology

For a given set of tasks the *general scheduling problem* asks for an order according to which the tasks are to be executed such that various constraints are satisfied. For a given set of real-time tasks, we want to devise a feasible allocation/schedule to satisfy timing constraints. The release time, the deadline and the execution time of the tasks are some of the parameters that should be considered when scheduling tasks on a real-

time system. The timing properties of a given task $\tau_j \in T = \{\tau_1, \tau_2, ..., \tau_n\}$ refer to the following [13, 17, 24, 7]:

- *Release time* (or *ready time* (r_j)): Time at which the task is ready for processing.
- *Deadline* (d_j): Time by which execution of the task should be completed.
- *Completion time* (C_j): Maximum time taken to complete the task, after the task is started.
- *Execution time* (e_j): Time taken without interruption to complete the task, after the task is started.
- *Priority* (ρ_j): Relative urgency of the task.
- *Period* (P_j): In the case of a periodic task τ_j, a period means once per a time interval of P_j or exactly P_j time units apart.
- *Utilization factor* (u_j): The utilization factor of a periodic task τ_j is defined by e_j/P_j. The utilization factor of a set of n periodic tasks is defined by $\sum_{i=1}^{n} e_i/P_i$.

Other issues to be considered in real-time scheduling are as follows.

- ***Periodic/Aperiodic/Sporadic tasks***

  Periodic real-time tasks are activated (released) regularly at fixed rates (periods). Aperiodic real-time tasks are activated irregularly at some unknown and possibly unbounded rate. The time constraint is usually a deadline. Sporadic real-time tasks are activated irregularly with some known bounded rate. The bounded rate is characterized by a minimum inter-arrival period, that is, a minimum interval of time between two successive activations. The time constraint is usually a deadline.

  A majority of sensory processing is periodic in nature. For example, a radar that tracks flights produces data at a fixed rate [16, 12, 11, 2].

- ***Preemptive/Non-preemptive tasks***

  In some real-time scheduling algorithms, a task can be preempted if another task of higher priority becomes ready. In contrast, the execution of a non-preemptive task should be completed without interruption once it has started [16, 13, 11, 2].

- ***Fixed/Dynamic priority scheduling***

  A priority is assigned to each task in priority driven scheduling. Assigning the priorities can be done statically or dynamically while the system is running [8, 24, 13, 16, 2].

  As for the second group of real-time scheduling problems, there exist polynomial algorithms which provide feasible schedules of any task set which satisfy some specific conditions. For example any set of periodic tasks which satisfy $\sum_{i=1}^{n} C_i/P_i \leq 1$ is guaranteed to be feasibly scheduled using EDF. Recall that an optimal scheduling algorithm is one which may fail to meet a deadline only if no other scheduling algorithm can meet the deadline. Therefore, a feasible scheduling algorithm is optimal if there is no other feasible algorithm with looser conditions. In order to prove the optimality of a scheduling algorithm, the feasibility conditions of the algorithm must be known. For example, there is no dynamic-priority scheduling algorithm that can successfully schedule a set of periodic tasks where $\sum_{i=1}^{n} C_i/P_i > 1$. Therefore, EDF is an optimal algorithm.

  The optimal algorithm for a real-time scheduling problem is not unique. For instance, in addition to the EDF algorithm, there is another optimal dynamic-priority scheduling algorithm, which is the least laxity first (LLF) algorithm.

- ***Multiprocessor/Single-processor systems***

  The number of available processors is one of the main factors in deciding how to schedule a real-time system. In multiprocessor real-time systems, the scheduling algorithms should prevent simultaneous access to shared resources and devices. Moreover, the best strategy to reduce the communication cost should be provided [16, 11].

  Multiprocessor scheduling techniques fall into two general categories:

  - *Global scheduling algorithms*

    Global scheduling algorithms store the tasks that have arrived, but not finished their execution, in one queue which is shared among all processors. Suppose there exist m processors. At every moment, the m highest priority tasks of the queue are selected for execution on the m processors using preemption and migration if necessary [16].

  - *Partitioning scheduling algorithms*

    Partitioning scheduling algorithms partition the set of tasks such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to

migrate, hence the multiprocessor scheduling problem is transformed to many uniprocessor scheduling problems [16].

For systems that contain more than one processor, we not only should decide about the appropriate scheduling algorithm, but also we have to specify the *allocation algorithm* which assigns the tasks to the available processors. For multiprocessor real-time systems, calculating the utilization bounds associated with *(scheduling, allocation)* algorithm pairs leads us to achieving the sufficient conditions for feasibly scheduling, analogous to those known for uniprocessors. This approach has several interesting features: it allows us to carry out fast schedulability tests and to qualify the influence of certain parameters, such as the number of processors, on scheduling. For some algorithms, this bound considers not only the number of processors, but also the number of the tasks and their sizes [8, 13, 16]. Global strategies have several disadvantages as compared with the partitioning strategies. Partitioning usually has a low scheduling overhead compared to global scheduling, because tasks do not need to migrate across processors. Furthermore, partitioning strategies reduce a multiprocessor scheduling problem to a set of uniprocessor ones and hence they allow the use of well-known uniprocessor scheduling algorithms for each processor. However, partitioning has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is an NP-complete problem. Thus, tasks are usually partitioned using non-optimal heuristics. Second, as shown in [3], task systems exist that are schedulable if and only if tasks are not partitioned. Still, partitioning approaches are widely used by system designers. In addition to the above approaches, we can apply hybrid partitioning/global strategies.

- ***Hyper period***

  Typically, if tasks $\tau_1, \tau_2, \tau_3, ..., \tau_n$ have periods $P_1, P_2, P_3, ..., P_n$, scheduling must be covered for a length of time equal to the *least common multiple* of the periods, i.e., $lcm(P_1, P_2, P_3, ..., P_n)$, as that is the time at which each task will have an integral number of invocations. If any of the P_i are co-primes, this length of time can be extremely large, so where possible it is advisable to choose values of P_i that are small multiples of a common value. We define a *hyper period* as the period equal to the least common multiple of the periods $P_1, P_2, P_3, ..., P_n$ of the n periodic tasks.

3 EDF scheduling

The EDF scheduling algorithm is a priority driven algorithm in which a higher priority is assigned to the request that has an earlier deadline, and a higher priority request always preempts a lower priority one [8, 13, 16, 11].

The following assumptions are made for the EDF algorithm.

(**a**) No task has any nonpreemptable section and the cost of preemption is negligible.

(**b**) Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.

(**c**) All tasks are independent; there are no precedence constraints.

(**d**) The tasks do not have to be periodic.

This scheduling algorithm in an example of priority driven algorithms with dynamic priority assignment in the sense that the priority of a request is assigned as the request arrives. Suppose each time a new ready task arrives, it is inserted into a queue of ready tasks which is sorted by the deadlines. If sorted lists are used, the EDF algorithm takes $O((N+\alpha)^2)$ time in the worst case, where N is the total number of the requests in each hyper period of n periodic tasks in the system and α is the number of aperiodic tasks.

EDF is an optimal uniprocessor scheduling algorithm. This can be proved by using a *time slice swapping* techniques. Using this technique, we can show that any valid schedule for any task set can be transformed into a valid EDF schedule [16].

It is shown in [16] that a set of periodic hard real-time tasks with relative deadlines equal to their periods, can be feasibly scheduled by the EDF scheduling algorithm on a uniprocessor system *if and only if* $\sum_{i=1}^{n} e_i/P_i \leq 1$ [16]. In Section 6, we discuss how to use this fact to solve the problem.

4 The Bin Packing Problem (BPP)

The Bin Packing problem can be defined as follows. Given N objects to be placed in bins of capacity V each, it is required to determine the minimum number of bins to accommodate all N objects. More formally, we are asked to find a partition and assignment of a set of objects such that a constraint is satisfied or an objective function is minimized. In this report, we take advantage of the heuristic algorithms provided for one-dimensional BPP.

In computational complexity theory, the bin packing problem is a combinatorial NP-hard problem, as observed in [1, 10, 15]. The Best Fit Decreasing (BFD) and First Fit Decreasing (FFD) algorithms are two approximation algorithms for the problem that use no more than $(11/9)OPT + 1$ bins, where OPT is the number of bins given by the optimal solution [4]. The simpler of these, the FFD strategy, operates by first sorting the items to be inserted in decreasing order by volume, and then by inserting each item into the first bin in the list with sufficient remaining space. The sorting step is relatively expensive, but without it for the Best Fit (BF) and First Fit (FF) algorithms, we only achieve the looser bound of $(17/10)OPT + 2$. A more efficient version of FFD uses no more than $(71/60)OPT + 1$ bins [9, 22].

5 Problem Definition

Consider a set of n hard real-time tasks $T = \{\tau_1, \tau_2, ..., \tau_n\}$. The tasks are periodic, independent and preemptive. Their deadlines are equal to their periods. Multiprocessing is acceptable. The processors are identical. Migration is not allowed. Except for the processors, there are no other shared resources in the system.

The objective is finding lower and upper bounds for the number of processors required by the following class of algorithms:

1 A partitioning strategy is used. Therefore we consider a combination of (allocation algorithm, scheduling algorithm) for any algorithm in this class.

2 The EDF scheduling algorithm is used on each processor.

3 Various allocation algorithms that are considered in this report share the following common properties: Start with one processor. Take tasks one by one and decide which of the existing processors should be assigned to the task, while sufficient conditions of EDF-schedulability should be satisfied for each processor. Each allocation algorithm adds a new processor to the system, only if there is not enough spare capacity for the new task on the existing processors to guarantee the feasibility of EDF-scheduling.

4 Considering items 1 and 2 above, it is guaranteed that all of the deadlines are met by any algorithm in this class. We prove this by contradiction as follows. Given an algorithm in the class, we assume that there exists at least one task that can not meet its deadline. In this case, there exists at least one processor which has been assigned to a set of tasks, including the task that can not meet its deadline, where

the EDF algorithm has not feasibly scheduled the set of tasks on the processor. But this is not possible, because, while assigning the processors to the tasks, the sufficient condition of feasibly scheduling by EDF is satisfied.

6 Lower and Upper Bounds and Heuristic Algorithms

Using the partitioning strategy, it is required to have an algorithm to allocate a processor to each task and a scheduling algorithm to schedule the tasks on each processor. In this report, we use either the BF or FF algorithm to determine the processor which should be assigned to each task. One might also use any other allocation algorithm which is suitable for the allocation algorithms discussed in Section 5. For any allocation algorithm in the defined class, a distinct priority is assigned to each task according to some mechanisms. This is why we sort the tasks and assign a higher priority to a task with higher index in the sorted array. After the priority assignment is made, then tasks are assigned to a number of processors based on an algorithm which we have chosen for allocation. After assignments have been done, for the tasks assigned to a given processor, a task with the earliest deadline receives the highest priority. It is the priority mechanism that is used in EFD scheduling algorithm. As mentioned earlier, we use the EDF scheduling algorithm for each processor, because employing the EDF algorithm allows the problem to be reducible to BPP with fixed size bins with capacity equal to one. In case we had used the RM scheduling algorithm, the problem would have been reduced to BBP with bins of various sizes. The EDF algorithm is an optimal uniprocessor scheduling algorithm, which is discussed in Section 3. Our problem will be reduced to finding an appropriate efficient algorithm that allocates the tasks to the minimum number of processors while meeting their deadlines is guaranteed. We know that so long as the sum of the utilizations of the tasks assigned to a processor is no greater than 1, the task set is EDF-schedulable on the processor. Therefore, the problem reduces to making task assignments with the property that the sum of the utilizations of the tasks assigned to a processor does not exceed 1. In this report, we study the allocation algorithms which pick the tasks one by one, verify EDF-schedulability to check if there is enough room for the task on one of the existing processors. If so, we assign the task to one of the processors, depending on the allocation algorithm, that has enough room. Otherwise, a new processor should be added to the system to accommodate the new task.

Our contributions in this report are as follows. We derive a lower bound for the minimum number of processors for any allocation algorithm that has the properties discussed earlier in this section. The lower bound that is obtained in this report is very tight and its results are very close to the results of the FFDU and BFDU algorithms (See

Section 6.1). The minimum number of the processors required for scheduling is equal to or larger than the proposed lower bound for the number of processors and smaller than the best heuristic algorithm. Since we look for the algorithm with the minimum number of processors, the best heuristic algorithm amongst the heuristic algorithms in Section 6.1 is the one with the least guaranteed performance. Simulation results lead to the fact that the difference between the number of processors achieved by the FFDU algorithm and the results of an optimal algorithm is negligible.

We also find an upper bound for the number of processors required by any allocation algorithm in our framework and with EDF-schedulability test on each processor.

Finally, we implement 12 heuristic algorithms with an EDF schedulability test. These algorithms are various combinations of a number of well-known algorithms. By implementing the heuristics, we aim to compare their simulation results with each other and with the upper and lower bounds derived in this report.

6.1 Heuristic Algorithms

We simulate the following heuristic algorithms. The heuristic scheduling algorithms can be categorized into two types of algorithms: the FF algorithms and the BF algorithms. In this work, we sort the tasks in increasing or decreasing order of execution time, period, or utilization of tasks.

- FFIE algorithm: sorts the tasks in ascending order of their execution times and then uses the FF algorithm with EDF-schedulability.
- FFIP algorithm: sorts the tasks in ascending order of their periods and then uses the FF algorithm with EDF-schedulability.
- FFIU algorithm: sorts the tasks in ascending order of their utilization factors and then uses the FF algorithm with EDF-schedulability.
- FFDE algorithm: sorts the tasks in descending order of their execution times and then uses the FF algorithm with EDF-schedulability.
- FFDP algorithm: sorts the tasks in descending order of their periods and then uses the FF algorithm with EDF-schedulability.
- FFDU algorithm: sorts the tasks in descending order of their utilization factors and then uses the FF algorithm with EDF-schedulability.
- BFIE algorithm: sorts the tasks in ascending order of their execution times and then uses the BF algorithm with EDF-schedulability.

- BFIP algorithm: sorts the tasks in ascending order of their periods and then uses the BF algorithm with EDF-schedulability.
- BFIU algorithm: sorts the tasks in ascending order of their utilization factors and then uses the BF algorithm with EDF-schedulability.
- BFDE algorithm: sorts the tasks in descending order of their execution times and then uses the BF algorithm with EDF-schedulability.
- BFDP algorithm: sorts the tasks in descending order of their periods and then uses the BF algorithm with EDF-schedulability.
- BFDU algorithm: sorts the tasks in descending order of their utilization factors and then uses the BF algorithm with EDF-schedulability.

The BF and FF algorithms are described as follows.

The BF Algorithm: In the BF algorithm, we pick the tasks one by one from the sorted array of the tasks, and then find the processor amongst the existing processors which has the largest remaining capacity. We then check whether the selected processor has enough room for the task in hand. If the chosen processor has enough room, we assign the task to the processor. Otherwise, none of the other existing processors have enough space, and we should add a new processor and assign the task to the new processor. We repeat the above procedure for all of the tasks one by one. We have provided six various algorithms of this type which are distinguishable by the order used to sort the tasks. The running time of the BF algorithms is $O(n\log n)$, where n is the number of hard real-time tasks.

The BF Algorithm with EDF-schedulability

1 *Sort the tasks based on either descending order or ascending order of one of the following parameters: execution time, period, or utilization factor.*

2 *Add one processor to the system and assign the first task to the processor*

3 *For all of the tasks,*

   - *Find the processor amongst the processors available which has the most remaining utilization capacity.*
   - *Check if the sum of the utilizations of the tasks assigned so far to the processor and the utilization of the new task exceeds 1.*

- *If not*
  Assign the task to the processor and add the utilization of the task to the total utilizations of the tasks so far assigned to the processor.
- *Otherwise, there is no processor among the existing processors that has enough capacity for the new task.*
  Add a new processor, assign the task to the new processor and set the total utilization of the tasks assigned to the processor as the utilization of the new task.

end if

end for

4 *Count the number of the processors used in the system.*

The FF Algorithm: Similar to the previous algorithm, we sort the tasks based on one of the above criteria into an array. We start with one processor. We pick the tasks one by one from the sorted array and assign them to the recently added processor as long as the utilization factor of the set of the tasks assigned to the processor does not exceed 1. Once it exceeds unity, we add another processor and assign the last task, which could not be assigned to the previous processor, to the new processor. We use the round robin method to pick the tasks. As the previous algorithm, we have provided six various of algorithms of this type which are distinguishable by the order used to sort the tasks. The running time of the FF algorithms are $O(n\log n)$, where n is the number of hard real-time tasks. In the following algorithm, V_j is the number of tasks that so far have verified if they can be assigned to processor j. Using V_j, we want to check if all of the unassigned tasks have been verified for possibility of assigning them to the j^{th} processor. TA is the total number of tasks that are assigned during the program. The tasks and processors are indexed by parameters i and j, respectively, in the following algorithm.

The FF Algorithm with EDF-schedulability

1 *Sort the tasks based on either descending order or ascending order of one of the following parameters: execution time, period, or utilization factor.*

2 *Let* $i = 1$.

3 *Add one processor to the system and assign the first task to the processor.*

4 *Set* $TA = 0$.

5 *Set* $V_j = 1$.

6 *While* $TA < n$,

- *If task* i *has not been assigned yet,*
  - *If there is enough room for task* i *on processor* j,
    - *Assign task* i *to processor* j.
    - $TA = TA + 1$.

    end if

  end if
- $i = i + 1$.
- *if* $i = (n + 1)$,
  - $i = 1$.

  end if
- $V_j = V_j + 1$.
- *If* $V_j = n$
  % in this case, none of the remaining unas-
  %signed tasks have the condition to be
  % assigned to the current processor.
  - *Add a new processor.*

  end if

end while

7 *Count the number of the processors used in the system.*

6.2 A Lower Bound

Theorem 1: *Consider a set of* n *periodic, independent, preemptive and hard real-time tasks* $T = \{\tau_1, \tau_2, ..., \tau_n\}$*, where their deadlines are equal to their periods. Multiprocessing is acceptable. The processors are identical. Migration is not allowed. Except for the processors, there are no other shared resources in the system. We have*

$$m \leq h,$$

where h is the number of the processors which are required to guarantee that the tasks can meet their deadlines by any algorithm, which belongs to the class of algorithms defined in Section 5, and $m = \lceil \sum_{i=1}^{n} (e_i/P_i) \rceil$, with e_i and P_i being the execution time and period of task τ_i.

Proof: We prove this theorem by contradiction. Assume that to the contrary the expression $m \nleq h$ is true.

In other words,

$$h < m \tag{1}$$

We consider any arbitrary set T_h of the tasks that are EDF-schedulable on h processors, when we use the partitioning strategy. Therefore, $T_h \subseteq T$, and since

$$\begin{array}{l} \sum_{\tau_j \text{is a task on the first processor}} (e_j/P_j) \leq 1 \\ \sum_{\tau_j \text{is a task on the second processor}} (e_j/P_j) \leq 1 \\ \sum_{\tau_j \text{is a task on the third processor}} (e_j/P_j) \leq 1 \\ \vdots \\ \sum_{\tau_j \text{is a task on the } h^{\text{th}} \text{ processor}} (e_j/P_j) \leq 1 \end{array}$$

we have

$$\sum_{\tau_j \in T_h} (e_j/P_j) \leq h \tag{2}$$

On the other hand, from the fact that h is a natural number and from (2), we conclude that

$$\left\lceil \sum_{\tau_j \in T_h} (e_j/P_j) \right\rceil \leq h \tag{3}$$

Considering $m = \lceil \sum_{i=1}^{n} (e_i/P_i) \rceil$ and from (1) and (3), we conclude that

$$\left\lceil \sum_{\tau_j \in T_h} (e_j/P_j) \right\rceil < \left\lceil \sum_{i=1}^{n} (e_i/P_i) \right\rceil \tag{4}$$

Therefore,

$$T_h \neq T. \tag{5}$$

From (5) and $T_h \subseteq T$, we conclude that $T_h \subset T$. Therefore, all of the tasks in T are not EDF-schedulable on h processors. Hence, $m \leq h$.

6.3 An Upper Bound

Theorem 2: *Consider a set of n periodic, independent, preemptive and hard real-time tasks* $T = \{\tau_1, \tau_2, ..., \tau_n\}$*, where their deadlines are equal to their periods. Multiprocessing is acceptable. The processors are identical. Migration is not allowed. Except for the processors, there are no other shared resources in the system. We have*

$$h < 2m,$$

where h is the number of the processors which are required to guarantee that the tasks can meet their deadlines by the algorithm, which belongs to the class of algorithms defined in Section 5, and $m = \lceil \sum_{i=1}^{n} (e_i/P_i) \rceil$*, where* e_i *and* P_i *are the execution time and period of a task* τ_i*, respectively.*

Proof: Suppose that the tasks have been assigned on h processors via one of the algorithms that satisfy the assumptions of the problem.

The occupied portion of the k^{th} processor is given by

$$U_k = \sum_{\tau_j \text{is a task on the } k^{\text{th}} \text{ processor}} \frac{e_j}{P_j} \tag{6}$$

and the unoccupied portion of the k^{th} processor is therefore

$$R_k = 1 - U_k. \tag{7}$$

We define R and U as follows.

$$R = \sum_{\tau_j \text{is a task assigned on one of the } h \text{ processors}} R_j \tag{8}$$

$$U = \sum_{\tau_j \text{is a task assigned on one of the } h \text{ processors}} U_j \tag{9}$$

From (6) and (9), we obtain

$$U = \sum_{i=1}^{n} (e_i/P_i). \tag{10}$$

Referring to the assumptions of the theorem, we have $m = \lceil \sum_{i=1}^{n} (e_i/P_i) \rceil$. Therefore, from (10), we obtain

$$m = \lceil U \rceil. \tag{11}$$

Having $U \leq \lceil U \rceil$ and from (11), we have the following inequality:

$$U \leq m. \tag{12}$$

On the other hand, from (7), (8) and (9), we conclude

$$R + U = h. \tag{13}$$

For any combination of (allocation algorithm, scheduling algorithm) of the class of algorithms considered in this report (refer to Section 6), we have

$$U_j > R_i, \tag{14}$$

where $j > i$. Otherwise, there was enough space for the tasks, which are currently assigned to the j^{th} processor, on the i^{th} processor and it was not required to add the j^{th} processor to the system.

From (14), we can say that $U_h > R_1$. Therefore, we have

$$1 - U_h < 1 - R_1. \tag{15}$$

Referring to (15) and (7), we obtain that

$$R_h < U_1. \tag{16}$$

From (16) and (14), we conclude the following set of inequalities:

$$\begin{cases} U_1 > R_h \\ U_2 > R_1 \\ U_3 > R_2 \\ \vdots \\ U_h > R_{h-1} \end{cases} \tag{17}$$

From (17), we conclude

$$R_1 + R_2 + \cdots + R_h < U_1 + U_2 + \cdots + U_h \tag{18}$$

Considering (8), (9) and (18), we have

$$R < U \tag{19}$$

From (19), (12), and (13), we conclude that $h < 2m$.

7 Simulation Results

We have implemented the 12 algorithms discussed in Section 6.1 to compare them with the upper and lower bounds provided in this report. Simulation conditions are as follows.

Each set of data includes n hard real-time tasks. For each task, we randomly generate e_i and p_i, which are the execution time and period, respectively. While generating the data sets, we have considered the relation $e_i < p_i$ for each task. Then, we generate 20 different data sets with size n. We next execute each of the aforementioned algorithms, namely the 12 heuristic algorithms and lower and upper bounds, on each data set. We compute the average of the aggregations of the number of the processors of the 20 simulations for data set with size n. The simulation is done for the algorithms for $n = 1$ to 500, with step 10.

Each heuristic algorithm allocates and schedules the tasks on a number of processors and computes the number of processors required by the algorithm. Lower and upper bounds are computed based on Theorem 1 and Theorem 2 derived in this report. Simulation results are provided in Figure 1. As expected, the results of the heuristic algorithms are between the upper and the lower bounds. We observe in Figure 1 that the FFDU and then BFDU algorithms have the closest outputs to the lower bound. The optimal solution lies between the lower bound and the FFDU. We also observe that the lower bound is very tight and the results of both of the FFDU and FDBU algorithms are very close to the optimal solution. For instance, it is shown in Figure 2 that for 20 sets of $n = 350$ tasks, the average of the lower bound for the required processors is equal to 133, where the FFDU and BFDU algorithms return 133.5 and 134 processors on the average, respectively. The optimal value is in the range of 133 to 134 processors. As we observe, the difference is negligible. In fact, the number of processors that are found by the lower bound are very close to the numbers found by the optimal algorithm. The running time of the lower bound is $O(n)$.

Moreover, the FFDU algorithm has polynomial, as opposed to exponential, growth in the number of tasks. In other words, it assigns a set of hard real-time tasks to a set of processors in polynomial time and the number of processors, which is required by this algorithm, is almost equal to minimum. As already stated, an efficient version of FFD uses no more than $(71/60)OPT + 1$ bins in BPP [9, 22]. This ratio states the guaranteed performance of the approximation algorithm. Comparing the numerical results of lower bound and the number of processors required by FFDU algorithm and considering the tightness of the lower bound, we observe that, on the average, the difference of performance of FFDU and the optimal algorithm is negligible.

Calculating the upper bound, we find out that by a combination of any allocation

algorithm (according to the definition given in this report) and the EDF scheduling algorithm, we require no more than $2\lceil\sum_{i=1}^{n}(e_i/P_i)\rceil$ processors to schedule the set of hard real-time tasks.

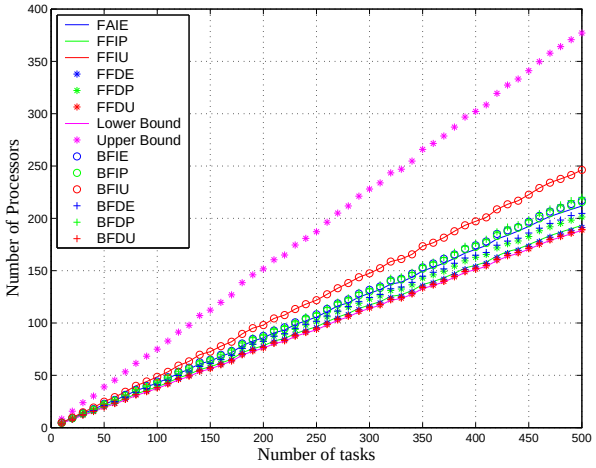

Figure 1: Number of processors required for scheduling n real-time tasks by various EDF-based partitioning algorithms

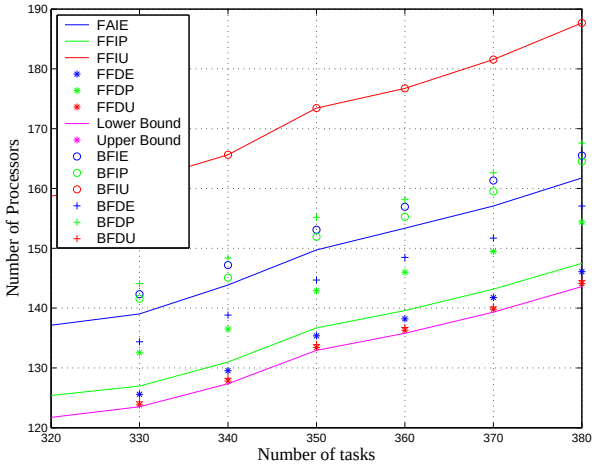

Figure 2: Enlarged version of a portion of Figure 1

8 Conclusions

In this work, we studied the problem of scheduling a set of independent periodic hard real-time tasks on the minimum number of identical processors such that the timing constraints of the events are satisfied. We focused on the partitioning strategy and when the EDF scheduling algorithm schedules the tasks assigned to each processor. Therefore, we looked for an appropriate allocation algorithm to assign the tasks to as few processors as possible, while the EDF-schedulablity test is satisfied on each processor. The problem is reducible to the BPP problem. Considering the fact that the problem is NP-hard, there is not any known polynomial algorithm to solve the problem. Since there exists some approximation heuristic algorithms for BPP, we can take advantage of the existing theorems for our problem.

In this report, we have derived an upper and a lower bound for the number of processors of any combination of (allocation, EDF scheduling) algorithm, where allocation algorithms are fit into the frame we have defined in Section 5. We implemented 12 heuristic algorithms and as we expected, the number of processors required by the heuristic algorithms lie between the bounds. Since we are looking for the algorithm that uses the minimum number of processors required, the optimal algorithm should be between the lower bound and the best approximation algorithm (the best algorithm among the 12 heuristic algorithms discussed in Section 4 are enough for our purposes). Among the 12 algorithms, the FFDU algorithm is the best and therefore, the optimal solution should be placed between the lower bound and the number of processors given by the FFDU algorithm. The lower bound is tight and the FFDU algorithm has a small guaranteed performance as compared with the optimal algorithm. As noted in the numerical results, the difference between the average number of processors from the FFDU algorithm and the lower bound is minimal. In fact, since the lower bound is tight, the difference between the results of the optimal algorithm and FFDU algorithm is negligible.

We have also derived an upper bound for the number of processors. This upper bound is observed to satisfy the properties described in Theorem 2.

References

[1] A. C. F. Alvim, F. Glover, C. C. Ribeiro and D. J. Aloise, "Local search for the bin packing problem," In Extended Abstracts of the III Metaheuristics International Conference (MIC99), Editted by P. Hansen and C. C. Ribeiro, pp. 7-12, Angra dos Reis, Brazil, 1999.

[2] G. C. Buttazzo, *"Hard Real-Time Computing Systems: predictable scheduling algorithms and applications,"* Springer, New York, NY, 2005.

[3] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, *"A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms,"* Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Edited by J. Y. Leung, CRC Press, Boca Raton, FL, USA, 2004.

[4] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein, "Approximation techniques for average completion time scheduling," *SIAM Journal on Computing*, Vol. 31, No. 1, pp. 146-166, 2001.

[5] Jr. Coffman, G. Galambos, S. Martello, and D. Vigo, "Bin Packing Approximation Algorithms: Combinatorial Analysis," in Handbook of Combinatorial Optimization, Editted by D. Du and P. Pardalos, Kluwer, Amsterdam, 1998.

[6] E. G. Coffman, Jr. M. R. Garey, and D. S. Johnson, "Approximation algorithms for NP-hard problems," Chapter Approximation algorithm for bin packing: A survey, PWS, pp. 46-93, Boston, MA, USA, 1996.

[7] W. Fornaciari, P. di Milano, *"Real Time Operating Systems Scheduling Lecturer,"* www.elet elet.polimi polimi.it/ fornacia it/ fornacia.

[8] K. Frazer, *"Real-time Operating System Scheduling Algorithms,"* http://home.earthlink.net/ krfrazer/Scheduler.pdf, 1997.

[9] M. R. Garey and D. S. Johnson, "A 71/60 theorem for bin packing," Journal of Complexity, Vol. 1, pp. 65106, 1985.

[10] M. R. Garey and D. S. Johnson, "Computers and Intractability : A Guide to the Theory of NP-Completeness," W. H. Freeman, January 15, 1979.

[11] W. A. Halang and A. D. Stoyenko, *"Real Time Computing,"* NATO ASI Series, Series F: Computer and Systems Sciences, Springer-Verlag, Vol. 127, New York, NY, USA, 1994.

[12] D. Isovic and G. Fohler, *"Efficient Scheduling of Sporadic, Aperiodic and Periodic Tasks with Complex Constraints,"* In Proceedings of the 21st IEEE RTSS, Florida, USA, November, 2000.

[13] M. Joseph, "Real-time Systems: Specification, Verification and Analysis," NATO ASI Series, Series F: Computer and Systems Sciences, Prentice Hall, NJ, 1996.

[14] C. -Y. Kao and F. -T. Lin, "A Stochastic Approach for the One-Dimensional Bin-Packing Problems," In Proceedings of 1992 IEEE International Conference on System, Man, and Cyberbetics, Chicago, Illinois, October 18-21, 1992.

[15] R. M. Karp, "Reducibility among combinatorial problems: in Complexity of Computer Computations," Editted by R. E. Miller and J. M. Thatcher, Plenum Press, pp. 85-103, New Your, NY, USA, 1972.

[16] C. M. Krishna and K. G. Shin, *"Real-Time Systems,"* MIT Press and McGraw-Hill, 1997.

[17] P. A. Laplante, "Real-time Systems Design and Analysis, An Engineer Handbook," IEEE Computer Society, IEEE Press, 1993.

[18] J. Leung and H. Zhao, "Real-Time Scheduling Analysis," Technical Report, Department of Computer Science, New Jersey Institute of Technology Newark, Newark, NJ, USA, November 2005.

[19] J. M. Lopez, M. Garca, J. L. Daz, and D. F. Garca, "Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling," Real Time Systems, Vol. 24, No. 1, January, 2003.

[20] J. M. Lopez, J. L. Daz, and D. F. Garca, "Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling," IEEE Transaction on Parallel and Distributed Systems, Vol. 15, No. 7, pp. 642-653, July, 2004.

[21] S. Martello and P. Toth, "Knapsack problems," John Wiley and Sons, 1990.

[22] Y. Minyi and Z. Lei, "A simple proof of the inequality MFFD$(L) = 71/60OPT(L)+$ 1, L for the MFFD bin-packing algorithm," Acta Mathematicae Applicatae Sinica 11, pp. 318330, 1995.

[23] A. Mohammadi and S. Akl, "Scheduling algorithms for real-time systems," Technical Report 2005-499, School of Computing, Queen's University, 2005.

[24] J. A. Stankovic and K. Ramamritham, "Tutorial Hard Real-Time Systems," IEEE Computer Society Press, 1988.