Partitioned Aperiodic Scheduling on Multiprocessors*

Björn Andersson† Tarek Abdelzaher‡ Jan Jonsson†

Abstract

This paper studies multiprocessor scheduling for aperiodic tasks where future arrivals are unknown. We propose an algorithm for tasks without migration capabilities and prove that it has a capacity bound of 0.31. No algorithm for tasks without migration capabilities can have a capacity bound greater than 0.50.

1 Introduction

In many applications, such as web servers, tasks arrive aperiodically, but we have no knowledge of future arrivals, so any task set could arrive causing the system to become overloaded and cause all tasks to miss their deadlines. Hence it is necessary to use an admission controller and the high frequency of these decisions requires the computational complexity of admission control be low. This efficiency in admission control can be achieved by ensuring that the load of admitted tasks is less than a number characteristic to the scheduling algorithm, its capacity bound.

Real-time applications often have inherent parallelism which results in sets of independent tasks, and hence they are suited for parallel processing. On certain platforms, such as networks-of-workstations, an aperiodic task may start to execute on any processor but once it has started to execute, task migration may be too costly: a non-migrative scheduling algorithm is needed. For periodically arriving tasks, there are several non-migrative scheduling algorithms available [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] but no algorithms for aperiodic scheduling with proven capacity bounds are currently available.

In this paper, we study multiprocessor scheduling algorithms and their capacity bounds for aperiodic tasks where future arrivals are unknown. We propose an algorithm for tasks without migration capabilities and prove that it has a capacity bound of 0.31. No algorithm for tasks without migration capabilities can have a capacity bound greater than 0.50.

Section 2 defines concepts and the system model that we use. Section 3 presents our new results: an algorithm to assign a task to a processor and the proof of its capacity bound. Section 4 closes the paper with conclusions and future work.

2 Concepts and System model

We consider the problem of scheduling a task set τ of aperiodically-arriving real-time tasks on m identical processors. A task τi has an arrival time Ai, an execution time Ci and a deadline Di, that is, the task requests to execute Ci time units during the time interval [Ai, Ai + Di). We assume that Ci and Di are positive real numbers such that Ci ≤ Di and Ai is a real number. With no loss of generality we can assume 0 = A1 ≤ A2 ≤ ... An. We let the set of current tasks at time t be defined as V(t) = {τk : Ak ≤ t < Ak + Dh}. For convenience, we define dh as dh = Ah + Dh.

The utilization ui of a task τi is ui = Ci/Di. The utilization at time t is U(t) = Στi∈V(t) ui. Since we consider scheduling on a multiprocessor system, the utilization is not always indicative of the load of the system because the original definition of utilization is a property of the current tasks only, and does not consider the number of processors. Therefore, we use the concept of system utilization, Us(t) = U(t)/m. The finishing time fi of a task τi is the earliest time when the task τi has executed Ci time units. If fi < Ai + Di, then we say that the task τi meets its deadline. We will analyze the performance of our scheduling algorithm using a capacity bound such that if the system utilization is, at every time, less than or equal to this capacity bound, then all deadlines are met. Our objective is to design a scheduling algorithm with a high capacity bound.

We study partitioned EDF (earliest deadline first) [11] and the system behaves as follows. When a task arrives, it is immediately assigned to a processor; a task is only allowed to execute on the processor to which it is assigned. (The reason why it is called partitioned scheduling is that at every moment the set of current tasks is partitioned so that tasks in one partition are assigned to one processor.) On each processor, the task with the highest priority of those tasks which have arrived, but not finished, is executed. The priorities of tasks are assigned according to EDF, that is, if di < dj then τi receives higher priority than τj and if di = dj then the priority ordering is arbitrary. The utilization of

---

*B This paper was funded in part by NSF grants CCR-0105873 and CCR-0206796, and by DARPA grant N00014-01-1-0576 and by the Swedish Foundation for Strategic Research via the national Swedish Real-Time Systems research initiative ARTES.

† Department of Computer Engineering, Chalmers University of Technology, SE–412 96 Göteborg, Sweden – {ba,janjo}@ce.chalmers.se.

‡ Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA – zaher@cs.virginia.edu.
processor \( p \) at time \( t \) is \( \sum_{\tau_i \in V_p} C_i / D_i \), where \( V_p = \{ \tau_k : (A_k \leq t < A_k + D_k) \land (\tau_k \text{ is assigned to processor } p) \} \).

A processor \( p \) is called occupied at time \( t \) if there is at least one task that is both current at time \( t \) and that is assigned to processor \( p \). A processor that is not occupied is called empty. Let \( \text{transition}_p(t) \) be the latest time \( \leq t \) such that processor \( p \) makes a transition from being free to being occupied at time \( \text{transition}_p(t) \). If a processor \( p \) has never been occupied, then \( \text{transition}_p(t) \) is \(-\infty\).

We assume that the scheduling algorithm\(^1\) is not allowed to use information about the future, that is, at time \( t \), it is not allowed to use \( A_i, D_i \) or \( C_i \) of tasks with \( A_i > t \). We say that a processor \( p \) is busy at time \( t \) if processor \( p \) executes some task, otherwise the processor \( p \) is idle at time \( t \). Note that an occupied processor is not necessarily busy. This could happen when all tasks have finished and hence they do not execute. But their deadlines have not expired so at least one processor is occupied. Analogously, a processor can be idle but the processor is not empty. Note that, every task that is assigned to processor \( p \) and is current at time \( t \) must have arrived during \( [\text{transition}_p(t), t] \). We will make use of this remark in a proof of our scheduling algorithm in Section 3.

We assume that tasks can always be preempted, and there is no cost of a preemption. A task can be assigned to any processor but it can only execute on the processor to which it was assigned when it arrived. Many tasks can be assigned to the same processor, but a processor cannot execute two or more tasks simultaneously. Tasks do not require exclusive access to any other resource than a processor.

### 3 Partitioned scheduling

In partitioned scheduling, a task is immediately assigned to a processor when the task arrives, and the task does not migrate, effectively making a multiprocessor behave as a set of uniprocessors. Algorithms that assign a task to a processor require knowledge of whether a task can be assigned to a processor and meet its deadline. We will make use of the following result\(^2\):\(^2\)

**Theorem 1** Consider EDF scheduling on a uniprocessor. If \( \forall t : U(t) \leq 1 \), then all tasks meet their deadlines.

**Proof:** Before proving this theorem, we will establish two claims:

1. If \( \forall t : U(t) \leq 1 \), then a uniprocessor sharing algorithm (called OPT) meets all deadlines.

This follows from the observation that a processor sharing algorithm attempts to execute an arbitrary active task \( \tau_i \) for \( u_i \cdot \epsilon \) time units during every time interval of length \( \epsilon \) within \( [A_i, A_i + D_i) \). Since \( \forall t : U(t) \leq 1 \), OPT succeeds to execute an arbitrary task \( \tau_i \) for \( u_i \cdot \epsilon \) time units during every time interval of length \( \epsilon \) within \( [A_i, A_i + D_i) \). One such interval is \( [A_i, A_i + D_i) \) and it has a length \( D_i \). In this interval, an arbitrary task \( \tau_i \) is executed for \( u_i \cdot \epsilon = u_i \cdot D_i = C_i \) time units. Hence OPT meets all deadlines.

2. If any scheduling algorithm meets all deadlines then EDF will also do so.

This follows from the optimality of EDF on a uniprocessor [13].

We can now reason as follows:

\[ \forall t : U(t) \leq 1 \quad \text{use claim 1} \]

\[ \text{OPT meets all deadlines} \quad \text{use claim 2} \]

\[ \text{EDF meets all deadlines} \]

\( \square \)

Intuitively, Theorem 1 reduces the partitioned multiprocessor scheduling problem to the design of an algorithm that assigns tasks to processors in order to keep the utilization on each processor at every moment to be no greater than 1.

When assigning tasks to processors, it is tempting to choose load balancing, but one can see that it can perform poorly in that it can miss deadlines even when only a small fraction of the capacity is requested. Example 1 illustrates that the capacity bound for load balancing is zero.

**Example 1** Consider \( m + 1 \) aperiodic tasks that should be scheduled on \( m \) processors using load balancing. We define load balancing as: assign an arriving task to a processor such that after the task has been assigned to a processor, the utilization of the processor that has the maximum processor utilization is minimized. Let the tasks \( \tau_i \) (where \( 1 \leq i \leq m \)) have \( D_i = 1, C_i = 2 \epsilon \), and \( A_i = i \cdot \epsilon \), and let the task \( \tau_{m+1} \) have \( D_{m+1} = 1 + \epsilon, C_{m+1} = 1 \) and \( A_{m+1} = (m + 1) \cdot \epsilon \). The tasks \( \tau_i \) (where \( 1 \leq i \leq m \)) will be assigned one processor each due to load balancing. When \( \tau_{m+1} \) arrives, it cannot be assigned to any processor to meet its deadline. By letting \( m \rightarrow \infty \) and \( \epsilon \rightarrow 0 \), we have a task set that requests an arbitrary small fraction of the capacity but still a deadline is missed.

In periodic scheduling, a common solution is to use binpacking algorithms [2]. Here, a task is first tentatively assigned to the processor with the lowest index, but if a schedulability test cannot guarantee that the task can be assigned there, then the task is tentatively assigned to the next processor with a higher index, and so on. This avoids the poor performance of load balancing [14, 10]. We will now apply these ideas in aperiodic scheduling by proposing a

---

1 By scheduling algorithm, we include the algorithm that assigns a task to a processor.

2 A more general theorem is available in [12].
new algorithm EDF-FF and analyzing its performance. Although the work by [14, 10] proved a capacity bound for the periodic case, their proof does not easily generalize because in our problem with aperiodic tasks, a task “disappears” when its deadline expires.

EDF-FF means schedule tasks according to Earliest-Deadline-First on each uniprocessor and assign tasks using First-Fit. EDF-FF works as follows. When a task \( \tau_i \) arrives it is assigned to the occupied processor with the lowest \( \text{transition}_{p}(A_i) \) that passes the schedulability condition of Theorem 1. Otherwise the task is assigned to an arbitrary empty processor (if no empty processor exists then EDF-FF declares failure). Because of Theorem 1, we know that if EDF-FF does not declare failure, then all deadlines are met. If two or more tasks arrive at the same time, there is a tie in that which task should be assigned first, and there could also be a tie in finding which processor is the one with the least transition. However, if there are tasks that arrive at the same time, then we can assign an ordering to them such that for every pair of these tasks, we can say that one task \( \tau_i \) arrives earlier than another \( \tau_j \). One such ordering could be to use the index of tasks. To illustrate this consider two tasks \( \tau_1 \) and \( \tau_2 \), with \( A_1 = 0 \) and \( A_2 = 0 \). The tasks have \( (C_1 = 0.6, D_1 = 1) \) and \( (C_1 = 0.7, D_2 = 1) \). It is clear that EDF-FF will not assign \( \tau_1 \) and \( \tau_2 \) to the same processor, but if we did not have a tie breaking scheme we would not know if EDF-FF would produce the assignment \( \tau_1 \) to processor 1 (and consequently \( \tau_2 \) to processor 2) or \( \tau_2 \) to processor 2 (and consequently \( \tau_1 \) to processor 1). Moreover, it would not be clear whether \( \text{transition}_1(t) < \text{transition}_2(t) \), for \( 0 < t < 0.6 \). To resolve this, we can choose an ordering so that \( A_1 \) is earlier than \( A_2 \), then \( \tau_1 \) is assigned to processor 1 and \( \tau_2 \) is assigned to processor 2 and \( \text{transition}_1(t) < \text{transition}_2(t) \) for \( 0 < t < 0.6 \). In the remainder of this section, if \( A_1 = A_j \), but it has been chosen that \( \tau_i \) arrives before \( \tau_j \), then we will write \( A_i < A_j \). The reason why this works is that the algorithm EDF-FF and its analysis does not depend on the absolute value of the arrival times; only the order is important.

Theorem 2 analyzes the performance of EDF-FF by computing its capacity bound \( B \). We will see that \( B \geq 0.31 \) and hence EDF-FF performs significantly better than load-balancing.

**Theorem 2** Consider scheduling on \( m \geq 3 \) processors using EDF-FF. Let \( B \) be a real number which is a solution to the equation

\[
m \cdot B = m \cdot (1 - B - B \cdot \ln \frac{m - 1}{B \cdot m - 1}) + \ln \frac{m - 1}{B \cdot m - 1}
\]

Then we know that:

1. There is exactly one solution \( B \), and it is in the interval \((1/m,1]\).
2. If \( \forall t: U(t) \leq m \cdot B \) then EDF-FF does not declare failure.

**Proof:**

1. There is exactly one solution \( B \), and it is in the interval \((1/m,1]\).

If \( B < 1/m \), then the right-hand side of Equation 1 has an imaginary part and then the left-hand side of Equation 1 has an imaginary part too, so \( B \) must have an imaginary part. But from the theorem we have that \( B \) is a real number, so this is not possible. Hence we have proven that \( B \geq 1/m \).

If \( B \to 1/m \), then the left-hand side is 1, and the right-hand side is \( m - 1 \), so this cannot be a solution. Hence we have proven that \( B > 1/m \).

Let us introduce the function \( f \) (which is just a manipulation of Equation 1):

\[
f(B) = m \cdot (1 - 2B - B \cdot \ln \frac{m - 1}{B \cdot m - 1}) + \ln \frac{m - 1}{B \cdot m - 1}
\]

We only need to prove that there is exactly one solution \( B \) to \( f(B) = 0 \). Noting that:

\[
\frac{\partial f}{\partial B} = -(1 + \ln \frac{m - 1}{B \cdot m - 1}) \cdot m < 0
\]

\[
\lim_{B \to 1/m} f(B) = m - 2 > 0
\]

\[
\lim_{B \to 1} f(B) = -m < 0
\]

makes it possible to draw the conclusion that there is exactly one solution, and it is in the interval \((1/m,1]\).

2. If \( \forall t: U(t) \leq m \cdot B \) then EDF-FF does not declare failure.

We will first derive a lower bound of the utilization of a task that was not assigned to one of the \( l \) occupied processors with the least \( \text{transition}_p \). Then we will assume that claim 2 in Theorem 2 was wrong, i.e. that there exist a task set with \( \forall t: U(t) \leq m \cdot B \) for which EDF-FF declares failure. We will use the result of the lower bound of the utilization of a task to prove a lower bound on the utilization at the time when EDF-FF declared failure. Finally this is used to derive a contradiction, which proves the correctness of claim 2 in Theorem 2.
A lower bound of the utilization of a task

Consider a task $\tau_i$ with utilization $u_i$ that arrives but is not assigned to the $l$ occupied processors with the least $\text{transition}_p(A_i)$. If one of the $l$ occupied processors had a utilization that was $1 - u_i$ or less, then $\tau_i$ would have been assigned to it but that did not happen. Consequently, every of the $l$ occupied processor with the least $\text{transition}_p(A_i)$ has a utilization greater than $1 - u_i$. Hence we have:

$$U(t = A_i) > l \cdot (1 - u_i) + u_i$$  \hspace{1cm} (2)

From the theorem we obtain:

$$U(t = A_i) \leq B \cdot m$$  \hspace{1cm} (3)

Combining Inequality 2 and Inequality 3 yields:

$$l \cdot (1 - u_i) + u_i < B \cdot m$$

Rearranging:

$$l - l \cdot u_i + u_i < B \cdot m$$

Rearranging again (for $l \geq 2$).

$$l - B \cdot m < (l - 1) \cdot u_i$$

Rearranging again (for $l \geq 2$).

$$\frac{l - B \cdot m}{l - 1} < u_i$$

We also know that $u_i > 0$. Hence we have (for $l \geq 2$):

$$\max(0, \frac{l - B \cdot m}{l - 1}) < u_i$$  \hspace{1cm} (4)

A lower bound of the utilization at failure

Suppose that claim 2 in Theorem 2 was wrong. Then there must exist a task set with $\forall t: U(t) \leq m \cdot B$ for which EDF-FF declares failure. Let $\tau_{\text{failed}}$ denote the task that arrived and caused the failure. If there was one processor that was empty at time $A_{\text{failed}}$, then $\tau_{\text{failed}}$ could have been assigned there and then EDF-FF would not have declared failure. For this reason, we know that all processors must have been occupied at time $A_{\text{failed}}$.

Let us choose the indices of processors so that $\text{transition}_1(A_{\text{failed}}) < \text{transition}_2(A_{\text{failed}}) < \ldots < \text{transition}_m(A_{\text{failed}})$. Every task that was current at time $A_{\text{failed}}$ and that was assigned processor $j$ must have arrived during $[\text{transition}_j(A_{\text{failed}}), A_{\text{failed}}]$ because processor $j$ was empty just before $\text{transition}_j(A_{\text{failed}})$, so any task that was current before $\text{transition}_j(A_{\text{failed}})$ and that was assigned to processor $j$ must have had its absolute deadline earlier than $\text{transition}_j(A_{\text{failed}})$. When a task $\tau_{\text{arrived}}$ arrived during $[\text{transition}_j(A_{\text{failed}}), A_{\text{failed}}]$ and was assigned to processor $j$, there were at least $j - 1$ occupied processors ($\text{processor}_1, \ldots , \text{processor}_{j-1}$) with a lower $\text{transition}_p(A_{\text{arrived}})$, so applying Inequality 4 (with $l = j - 1$) gives (for $j \geq 3$):

$$\max(0, \frac{j - 1 - B \cdot m}{j - 2}) < u_{\text{arrived}}$$  \hspace{1cm} (5)

Since all processors are occupied at time $A_{\text{failed}}$, for every processor $j \in [1..m]$, there is at least one current task assigned to processor $j$ that satisfies Inequality 5. For this reason (for $j \geq 3$) the utilization of processor $j$ at time $A_{\text{failed}}$ must satisfy:

$$U_j(t = A_{\text{failed}}) > \max(0, \frac{j - 1 - B \cdot m}{j - 2})$$  \hspace{1cm} (6)

where $U_j(t = A_{\text{failed}})$ denotes the utilization of processor $j$ at time $A_{\text{failed}}$.

We also know that the task $\tau_{\text{failed}}$ was not assigned to the $m$ occupied processors with the least $\text{transition}_p(A_{\text{failed}})$, so applying Inequality 4 with $l = m$ gives:

$$u_{\text{failed}}(t = A_{\text{failed}}) > \max(0, \frac{m - B \cdot m}{m - 1})$$  \hspace{1cm} (7)

When EDF-FF failed, the utilization of all current tasks is the same as the utilization of all current tasks that has been assigned processors plus the utilization of the task that just arrived. Hence:

$$U(t = A_{\text{failed}}) = \left( \sum_{j=1}^{m} U_j(t = A_{\text{failed}}) \right) + u_{\text{failed}}$$

Applying Inequality 6 and Inequality 7 and using $\sum_{j=1}^{m} U_j(t = A_{\text{failed}}) > \sum_{j=3}^{m} U_j(t = A_{\text{failed}})$ yields:

$$U(t = A_{\text{failed}}) > \left( \sum_{j=3}^{m} \max(0, \frac{(j - 1 - B \cdot m)}{(j - 2)}) \right)$$

$$+ \max(0, \frac{m - B \cdot m}{m - 1})$$  \hspace{1cm} (8)

Rewriting (see Algebraic manipulation 1 in the Appendix for details) gives us:

$$U(t = A_{\text{failed}}) > m - 1 - \sum_{h=1}^{m-1} \min(1, \frac{B \cdot m - 1}{h})$$  \hspace{1cm} (9)
It is worth to notice that every term in the sum of Inequality 9 is non-negative because from the first claim in the theorem, we have $B > 1/m$, which can be rewritten to $B \cdot m - 1 > 0$. We will now compute an upper bound on $\sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k})$. Clearly we have:

$$\sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) = \sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) + \sum_{k=2}^{m-1} \min(1, \frac{B \cdot m - 1}{k})$$

Observing that the series: $\min(1, \frac{B \cdot m - 1}{k})$ is non-increasing with respect to $k$ and that $\sum_{k=1}^{1} \min(1, \frac{B \cdot m - 1}{k}) = \min(1, \frac{B \cdot m - 1}{1}) \leq 1$ yields:

$$\sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) < 1 + \int_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k})$$

Rewriting (see Algebraic manipulation 2 in the Appendix for details) gives us:

$$\sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \leq 1 + B \cdot (m - 1 - 1) + (B \cdot m - 1) \cdot \ln \frac{m - 1}{B \cdot m - 1} \quad (10)$$

Using Inequality 10 in Inequality 9 yields:

$$U(t = A_{failed}) > m - 1 - (1 + B \cdot m - 1 - 1) + (B \cdot m - 1) \cdot \ln \frac{m - 1}{B \cdot m - 1}$$

Simplifying yields:

$$U(t = A_{failed}) > m - B \cdot m - B \cdot m - 1 + \ln \frac{m - 1}{B \cdot m - 1}$$

Simplifying again:

$$U(t = A_{failed}) > m \cdot (1 - B - B \cdot \ln \frac{m - 1}{B \cdot m - 1}) + \ln \frac{m - 1}{B \cdot m - 1}$$

Since $U(t = A_{failed}) \leq m \cdot B$ it must have been that $m \cdot B > m \cdot (1 - B - B \cdot \ln \frac{m - 1}{B \cdot m - 1}) + \ln \frac{m - 1}{B \cdot m - 1}$

But this is impossible, because we have chosen $B$ such that $m \cdot B = m \cdot (1 - B - B \cdot \ln \frac{m - 1}{B \cdot m - 1}) + \ln \frac{m - 1}{B \cdot m - 1}$.

Different values of $B$ are shown in Table 1. When $m$ approaches infinity, then $B$ is the solution to $1 - 2B + 2B \ln B = 0$. This is where our capacity bound of 0.31 came from.

Our analysis of capacity bounds of EDF-FF is not necessarily tight but one can see that no analysis of capacity bounds of EDF-FF can, in general, obtain a capacity bound that is the greater than 0.42. This is illustrated in Example 2.

**Example 2 (Adapted from [15])** Let $m = 7k/3$, where $k$ is divisible by 6. First $k$ tasks with $u_i = 2/3 - \epsilon$ arrive followed by $k$ tasks with $u_i = 1/3 - \epsilon$ and then $k$ tasks with $u_i = 2\epsilon$. Figure 1(a) illustrates the packing of current tasks.

The deadline of tasks with $u_i = 1/3 - \epsilon$ expires, so we remove them. $k/2$ tasks with $u_i = 1/3$ and $k/2$ tasks with $u_i = 1/3 + \epsilon$ arrive in the sequence $1/3, 1/3 + \epsilon, 1/3, 1/3 + \epsilon, \ldots, 1/3, 1/3 + \epsilon$. Figure 1(b) illustrates the packing of current tasks.

The deadline of tasks with $u_i = 2/3 - \epsilon$ expires, so we remove them. The deadline of tasks with $u_i = 1/3 + \epsilon$ expires, so we remove them too. $5k/6$ tasks with $u_i = 1$ arrive. Now, there is at least one task assigned to every processor. Figure 1(c) illustrates the packing of current tasks. Finally, a task with $u_i = 1/3$ arrives, but it cannot be assigned to any processor, so EDF-FF fails. If we let $\epsilon \to 0$ and $k \to \infty$ (and consequently $m \to \infty$), then we have that $\forall t: U(t) \leq 3/7$, but EDF-FF still fails.

Although, one could design other partitioning schemes, those are unlikely to offer any significant performance improvements, because in dynamic bin-packing (where items arrive and depart) it is known that first-fit offers a performance (in terms of competitive ratio) not too far from the best that one could hope for [15, page 230]. Nevertheless, it is known [10] that no partitioned multiprocessor scheduling algorithm can achieve a capacity bound greater than 0.5.

For computer platforms where task migration is prohibitively expensive, one could conceive of other approaches than partitioning. For example, one could conceive of scheduling algorithms where an arriving task is assigned to a global runnable queue, and when it has started to execute, it is assigned to that processor and does not migrate. However, such approaches suffer from scheduling anomalies [16] and they still cannot achieve a capacity bound greater than 0.50 (to see this consider the same example as given in [10]).
Table 1: B for different number of processors.

<table>
<thead>
<tr>
<th>m</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.41</td>
<td>0.38</td>
<td>0.36</td>
<td>0.35</td>
<td>0.34</td>
<td>0.32</td>
<td>0.31</td>
<td>0.31</td>
</tr>
</tbody>
</table>

We conclude that although it may be possible to design and analyze scheduling algorithms that do not migrate tasks and make these algorithms achieve greater capacity bounds, these algorithms will not improve the performance as much as the EDF-FF did (from 0 to 0.31).

4 Conclusion and future work

This paper has studied multiprocessor scheduling for aperiodic tasks where future arrivals are unknown. We proposed a priority-driven algorithm without task migration and proved that it has a capacity bound of 0.31. No previous work has proven a capacity bound for aperiodic multiprocessor scheduling for non-migrative tasks. We also saw that no algorithm for tasks without migration capabilities can have a capacity bound greater than 0.50. A possible future work is the design of a scheduling algorithm with the same utilization bound as the one presented here but with shorter average response times. This may be possible by assigning a task to an empty processor when the number of occupied processors are small enough.

Acknowledgement

The authors are grateful to helpful comments from one anonymous reviewer.

References


(a) Tasks arrive and are assigned to the processors $P_1, \ldots, P_k$ but the deadlines of tasks have not expired yet.

(b) The deadlines of tasks with utilization $1/3 - \epsilon$ expire and new tasks arrive and are assigned to processors $P_{k+1}, \ldots, P_{3k/2}$.

(c) The deadlines of tasks expire and tasks with utilization one arrive and are assigned to processors $P_{3k/2+1}, \ldots, P_{7k/3}$.

Figure 1: No analysis of EDF-FF can give a capacity bounds greater than 0.42.


A Algebraic manipulations

Algebraic manipulation 1 Consider:

$$U(t = A_{\text{failed}}) > \sum_{j=3}^{m} \max(0, \frac{(j - 1) - B \cdot m}{(j - 2)}) + \max(0, \frac{m - B \cdot m}{m - 1})$$

Rewriting yields:

$$U(t = A_{\text{failed}}) > \sum_{j=3}^{m+1} \max(0, \frac{(j - 1) - B \cdot m}{(j - 2)})$$

Substitute $k = j - 2$ yields:

$$U(t = A_{\text{failed}}) > \sum_{k=1}^{m-1} \max(0, \frac{(k + 1) - B \cdot m}{k})$$

Rearranging:

$$U(t = A_{\text{failed}}) > \sum_{k=1}^{m-1} \max(0, 1 - \frac{B \cdot m}{k})$$

Rearranging:

$$U(t = A_{\text{failed}}) > \sum_{k=1}^{m-1} \max(0, 1 - \frac{B \cdot m - 1}{k})$$

Rearranging:

$$U(t = A_{\text{failed}}) > \sum_{k=1}^{m-1} \max(0, 1 - \frac{B \cdot m - 1}{k})$$
Rearranging:

\[ U(t = A_{\text{faultd}}) > m - 1 - \sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \]

**Algebraic manipulation 2** Consider:

\[ \sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \leq 1 + \int_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \]  
\[ (11) \]

We can rewrite the integral to:

\[ \int_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) = \int_{k=1}^{B \cdot m - 1} \min(1, \frac{B \cdot m - 1}{k}) + \int_{k=B \cdot m - 1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \]

Rewriting again:

\[ \int_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) = \int_{k=1}^{B \cdot m - 1} 1 + \int_{k=B \cdot m - 1}^{m-1} \frac{B \cdot m - 1}{k} \]

Rewriting again:

\[ \int_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) = B \cdot m - 1 - 1 + (B \cdot m - 1) \cdot \ln \frac{m - 1}{B \cdot m - 1} \]  
\[ (12) \]

Substituting Equation 12 in Inequality 11 yields:

\[ \sum_{k=1}^{m-1} \min(1, \frac{B \cdot m - 1}{k}) \leq 1 + B \cdot m - 1 - 1 + (B \cdot m - 1) \cdot \ln \frac{m - 1}{B \cdot m - 1} \]