Global Priority-Driven Aperiodic Scheduling on Multiprocessors

Björn Andersson† Tarek Abdelzaher‡ Jan Jonsson†

Abstract

This paper studies multiprocessor scheduling for aperiodic tasks where future arrivals are unknown. A previously proposed priority-driven scheduling algorithm for periodic tasks with migration capability is extended to aperiodic scheduling and is shown to have a capacity bound of 0.5. This bound is close to the best achievable for a priority-driven scheduling algorithm. With an infinite number of processors, no priority-driven scheduling algorithm can perform better. We also propose a simple admission controller which guarantees that admitted tasks meet their deadlines and for many workloads, it admits tasks so that the utilization can be kept above the capacity bound.

1 Introduction

In many applications, such as web servers, tasks arrive aperiodically. In other applications, tasks arrive recurrently, but treating them as unrelated aperiodic tasks enables fine-grained run-time QoS adaptations [1]. For example, a task arrives because an event occurred or a sensor reading is available, and at that time (but not earlier) accurate estimates on the execution time of tasks can be computed from the input data [2]. These estimates are close to the true ones [3], so using them in scheduling and admission control can keep processor utilization high while meeting all deadlines, despite highly variable execution times. The high frequency of these decisions require 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, called the capacity bound.

Real-time applications often have inherent parallelism which results in sets of independent tasks, and hence they are suited for parallel processing. For certain architectures, such as uniform memory-access shared-memory multiprocessors, the cost of task migration is low enough to allow global scheduling, that is, a task is allowed to execute on any processor even when resuming after having been preempted. Global scheduling offers short average response-times, ability to meet deadlines despite variable execution times [4] and it can consider a simultaneously multithreaded processor as a multiprocessor while offering good performance [5]. For periodically arriving tasks, there are several global scheduling algorithms available (see for example [6, 7, 8]) but algorithms for aperiodic scheduling with proven capacity bounds are currently only available for a uniprocessor [9] or for a restricted task model in multiprocessors [10].

In this paper, we study global multiprocessor scheduling algorithms and their capacity bounds for aperiodic tasks where future arrivals are unknown. In particular, we extend a previously proposed priority-driven scheduling algorithm for periodic tasks with migration capability to aperiodic scheduling and show that it has a capacity bound of 0.5. This bound is close to the best achievable for a priority-driven scheduling algorithm. With an infinite number of processors, no priority-driven scheduling algorithm can perform better. We also propose a simple admission controller which guarantees that admitted tasks meet their deadlines and for many workloads, it admits tasks so that the utilization can be kept above the capacity bound.

The remainder of this paper is organized as follows. Section 2 defines concepts and the system model that we use. Section 3 describes ideas in the design of global multiprocessor scheduling algorithms. Section 4 presents the algorithm EDF-US(m/(2m-1)) and Section 5 presents the derivation of its capacity bound. Section 6 presents the new admission controller and evaluates its performance. Section 7 closes the paper with conclusions and future work.

2 Concepts and System model

We consider the problem of scheduling a task set $\tau$ of aperiodically-arriving real-time tasks on $m$ identical processors. A task $\tau_i$ has an arrival time $A_i$, an execution time $C_i$ and a deadline $D_i$, that is, the task requests to execute $C_i$ time units during the time interval $[A_i, A_i + D_i]$. We assume that $C_i$ and $D_i$ are positive real numbers such that $C_i \leq D_i$ and $A_i$ is a real number. With no loss of generality we can assume $0 = A_1 \leq A_2 \leq \ldots A_m$. We let the set of current tasks at time $t$ be defined as $V(t) = \{\tau_k : A_k \leq t < A_k + D_k\}$. For convenience, we call $A_k + D_k$ the absolute deadline of the task $\tau_k$ and call $D_k$ the relative deadline of...
The utilization \( u_t \) of a task \( \tau_i \) is \( u_t = C_t / D_t \). The utilization at time \( t \) is \( U(t) = -\sum_{\tau \in V(t)} u_{\tau} \). 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, \( U_s(t) = U(t) / m \). The finishing time \( f_t \) of a task \( \tau_i \) is the earliest time when the task \( \tau_i \) has executed \( C_t \) time units. If \( f_t \leq A_i + D_i \), then we say that the task \( \tau_i \) 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.

In periodic scheduling, it is common to distinguish between dynamic- and static-priority scheduling, in that static-priority scheduling requires all arrivals of a task to have the same priority. However in our model of aperiodic scheduling, a task only arrives once, making such a definition meaningless, so we will, inspired by Ha and Liu [11], study priority-driven scheduling, defined as follows. A priority-driven scheduling algorithm assigns priorities to tasks, so that for every pair of tasks, one task has higher priority than the other task in the pair, and these relative priority orderings never change. DM (deadline monotonic [12]) and EDF (earliest deadline first [13]) are priority-driven, whereas PF [6] is not priority-driven.

We study global priority-driven scheduling and the system behaves as follows. Each task is assigned a global priority. Of all tasks that have arrived, but not finished, the \( m \) highest-priority tasks are executed in parallel on the \( m \) processors.

We assume that the scheduling algorithm is not allowed to use information about the future, that is, at time \( t \), it is not allowed to use \( A_t \), \( D_t \) or \( C_t \) of tasks with \( A_t > t \). We assume that tasks can always be preempted, and there is no cost of a preemption, even if the task resumed on another processor. A task cannot execute on two or more processors simultaneously and a processor cannot execute two or more tasks simultaneously. Tasks do not require exclusive access to any other resource than a processor.

### 3 Global priority-driven scheduling

In global preemptive priority-driven scheduling, the schedule is completely defined by the number of processors and the arrival times, deadlines, execution times and priorities of tasks, so the scheduling problem amounts to assigning priorities to tasks. Clearly a priority assignment scheme must consider the relative or absolute deadlines of tasks; a natural choice is to use DM2 or EDF3. This works well on a multiprocessor when the utilizations of all tasks are small [10, 14]. However, if the utilization of a task can be large, algorithms such as DM and EDF can miss deadlines even if the system utilization is always arbitrarily close to zero (to see this, consider the task set in Example 1).

#### Example 1

Consider \( m + 1 \) aperiodic tasks that should be scheduled on \( m \) processors using deadline monotonic scheduling or earliest deadline first scheduling. Let the tasks \( \tau_i \) (where \( 1 \leq i \leq m \)) have \( D_t = 1, C_t = 2\epsilon \) and \( A_t = i \cdot \epsilon^2 \), and let the task \( \tau_{m+1} \) have \( D_{m+1} = 1 + \epsilon, C_{m+1} = 1 \) and \( A_{m+1} = (m + 1) \cdot \epsilon^2 \). The tasks \( \tau_i \) execute immediately when they arrive, are not preempted and hence meet their deadlines. When \( \tau_{m+1} \) arrives, it receives the lowest priority so it is not given enough contiguous time and hence it misses its deadline.

By letting \( m \to \infty \) and \( \epsilon \to 0 \), we have a task set that requests an arbitrary small fraction of the capacity but still a deadline is missed.

The reason for the undesirable behavior in Example 1 is that there could be plenty of time intervals on the processors that could be used for a task’s execution, but these time intervals are distributed on different processors and overlap in time. Since a task cannot execute on two or more processors simultaneously, a large fraction of these time intervals cannot be used. This problem becomes more severe when tasks with long execution times receive the lowest priority. For example, a task needs to execute \( C_t \) time units during \([A_t, A_t + D_t] \), and during this time interval the higher priority tasks execute in such a way that all processors are available for executing the lowest priority task at the same time during a time interval of length \( C_t - \epsilon \). In this case, task \( \tau_i \) misses its deadline although there is in total \((C_t - \epsilon) \cdot m \) units of time available. To counter this, researchers have designed priority-assignment schemes that give a high priority to tasks with a long execution time (or utilization) but also give high priority to tasks with a short deadline [7, 8, 15].

EDF-US(m/(2m-1)) [8] is a recently-proposed priority-driven scheduling multiprocessor algorithm which is based on this idea. This is the priority-driven scheduling algorithm with the highest capacity bound. Unfortunately, the algorithm was only designed and analyzed for periodic scheduling. For this reason, we will extend EDF-US(m/(2m-1)) and analyze it for aperiodic tasks. In fact, the old design and analysis of EDF-US(m/(2m-1)) turns out as a special case of our EDF-US(m/(2m-1)). Section 4 describes our EDF-US(m/(2m-1)) in aperiodic scheduling and

---

1. At each instant, the processor chosen for each of the \( m \) tasks is arbitrary. If less than \( m \) tasks should be executed simultaneously, some processors will be idle.
2. This means deadline monotonic and it gives the highest priority to the task with the least \( D_t \).
3. This means earliest deadline first and it gives the highest priority to the task with the least absolute deadline.
Section 5 analyzes the performance of our EDF-US(m/(2m-1)) in aperiodic scheduling.

4 The design of EDF-US(m/(2m-1))

EDF-US(m/(2m-1)) means Earliest-Deadline-First Utilization-Separation with separator m/(2m-1). We say that a task \( \tau \) is heavy if \( u_\tau > m/(2m-1) \) and a task is light if \( u_\tau \leq m/(2m-1) \). EDF-US(m/(2m-1)) assigns priorities so that all heavy tasks receive higher priority than all light tasks. The relative priority ordering among the heavy tasks are arbitrary, but the relative priority ordering among the light tasks is given by EDF. The rationale for this separation of heavy and light tasks (as suggested earlier in Section 3) is that if heavy tasks would have received a low priority, then heavy tasks could miss deadlines even if there is ample of capacity available.

5 The capacity bound of EDF-US(m/(2m-1))

In order to derive a capacity bound for EDF-US(m/(2m-1)) when used with aperiodic tasks, we will look at the case with periodic tasks [8]. There, it was shown that if all tasks are light and the system utilization is always no greater than \( m/(2m-1) \), then EDF schedules all tasks to meet their deadlines. If every heavy task was assigned its own processor and the light tasks executed on the remaining processors and the system utilization of all these tasks is no greater than \( m/(2m-1) \), then all deadlines would also hold. It was also shown that even if a light task would be allowed to execute on a processor where a heavy task executes when the heavy tasks does not execute, then deadlines continue to hold. The reason why this technique works for periodic tasks is that the number of current tasks never changes at run-time because when a deadline of a current task has expired, a new current task with the same execution time and deadline arrives.

However, in aperiodic scheduling, the number of heavy tasks is not necessarily the same at all times. Hence, the number of processors that are available for light tasks may vary with time. For this reason, we will prove (in Lemma 4) a schedulability condition of EDF for light tasks when the number of processors varies. We will do this in two steps. First, we will prove that OPT, an optimal scheduling algorithm, meets deadlines. Second, we will prove (in Lemma 3) that if any scheduling algorithm meets deadlines, then EDF will also do so if EDF is provided faster processors. The second step is proven by using a result (Lemma 2) that tells how much work a scheduling algorithm does. To do this, we need to define work, OPT and a few other concepts.

Since we study scheduling on identical processors, all processors in a computer system have the same speed, denoted \( s \), but two different computer systems may have processors of different speed. If the speed of a processor is not explicitly written out, it is assumed that \( s = 1 \). A processor that is busy executing tasks during a time interval of length \( l \) does \( l \cdot s \) units of work. This means that if a processor of speed \( s \) starts to execute a task with execution time \( t \) at time 0, then the task has finished its execution at time \( t + l \). Let \( W(\tau, m(t), s, \tau, t) \) denote the amount of work done by the task set \( \tau \) during the time interval \([0, t]\) scheduled by algorithm \( A \) when the number of processors varies according to \( m(t) \) and each processor runs with the speed \( s \). We assume that \( m(t) \) changes at the time instants denoted \( t_1, t_2, \ldots \) and let \( m_{UB} \) be a number such that \( \forall t: m(t) \leq m_{UB} \). For convenience, we will say that a computer system has \( m(t) \) processors when we mean that the number of processors varies as a function \( m(t) \), where \( t \) is time.

Let OPT denote an algorithm that executes every current task \( \tau_i, L \cdot C_i/D_i \) time units in every time interval of length \( L \). It implies that a task \( \tau_i \) will execute \( C_i \) time units in every time interval of length \( D_i \). In particular, it implies that \( \tau_i \) will execute \( C_i \) time units in the interval \([A_i, A_i + D_i]\), and hence it meets its deadline. One can see that if \( \forall t: U(t) \leq m(t) \) then OPT will succeed to schedule every current task \( \tau_i, L \cdot C_i/D_i \) time units in every time interval of length \( L \) and hence they meet their deadlines. We first show that there exists an algorithm OPT that has these properties and that never executes a task on two or more processors simultaneously. To that end, we can make use of Theorem 1 in [16], which we repeat below, and for convenience have rewritten to use our notation and made a trivial rewriting.

**Lemma 1** If there are \( m \) processors and \( n \) tasks, with all \( A_i = 0, D_i = K > 0 \), and if preemption is allowed, but no task may be processed on two machines simultaneously, then there exists a schedule which finishes all tasks by \( K \) if, and only if

\[
\begin{align*}
(1) & \quad \frac{C_i}{D_i} \leq 1 \text{ for each task } \tau_i \text{ and} \\
(2) & \quad \sum_{i=1}^{m} \frac{C_i}{D_i} \leq m
\end{align*}
\]

**Proof:** See [16].

We can split an arbitrary time interval of length \( L \) into a set \( \{[s_1, e_1], [s_2, e_2], \ldots, [s_n, e_n]\} \) of time intervals with \( s_j+1 = e_j \) such that in each time interval \([s_j, e_j]\) the number of processors does not change and the number of current tasks does not change. We also split a task \( \tau_i \) into \( l_i \) subtasks \( \tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,l_i} \), such that \( \tau_{i,j} \) have \( A_{i,j} = s_j \cdot D_{i,j} = e_j - s_j \) and \( C_{i,j} = (e_j - s_j) \cdot \frac{C_i}{D_i} \). Since Lemma 1 assures us that each subtask meets its deadline, then it holds that every task meets their deadlines. We conclude that there is an algorithm OPT that never executes a task on two or more processors simultaneously and it can guarantee that if \( \forall t: U(t) \leq m(t) \) then all deadlines are met.

Later when we prove the condition of schedulability of EDF, we will need the following lemma that tells us how
much work a work-conserving scheduling algorithm performs compared to any scheduling algorithm. Such a condition was proven by [17] but it assumed that the number of processors did not vary; we will remove this limitation. We say that a scheduling algorithm is work-conserving if it never leaves a processor idle when there are tasks available for execution. For our purposes, EDF-US(m(2m-1)) is work-conserving.

**Lemma 2** Consider scheduling on \( m(t) \) processors. Let \( A \) be an arbitrary work-conserving scheduling algorithm and let \( A' \) be an arbitrary scheduling algorithm. Then we have:

\[
W(A, m(t), (2 - \frac{1}{m_{UB}}) \cdot s, \tau, t) \geq W(A', m(t), s, \tau, t)
\]

**Proof:** The proof is by contradiction. Suppose that it is not true; i.e., there is some time-instant by which a work-conserving algorithm \( A \) executing on \( (2 - \frac{1}{m_{UB}}) \cdot s \)-speed processors has performed strictly less work than some other algorithm \( A' \) executing on \( s \)-speed processors.

Let \( \tau_j \in T \) denote a task with the earliest arrival time such that there is some time-instant \( t_0 \) satisfying

\[
W(A, m(t), (2 - \frac{1}{m_{UB}}) \cdot s, \tau, t_0) < W(A', m(t), s, \tau, t_0)
\]

and the amount of work done on task \( \tau_j \) by time-instant \( t_0 \) in \( A \) is strictly less than the amount of work done of \( \tau_j \) by time-instant \( t_0 \) in \( A' \). One such \( \tau_j \) must exist, because there is a time \( t < t_0 \) such that \( W(A, m(t), (2 - \frac{1}{m_{UB}}) \cdot s, \tau, t) = W(A', m(t), s, \tau, t) \). For example, \( t = 0 \) gives one such equality. By our choice of \( A_j \), it must be the case that

\[
W(A, m(t), (2 - \frac{1}{m_{UB}}) \cdot s, \tau, A_j) \geq W(A', m, s, \tau, A_j)
\]

Therefore, the amount of work done by \( A' \) over \([A_j, t_0)\) is strictly more than the amount of work done by \( A \) over the same interval. The fact that the amount of work done on \( \tau_j \) in \([A_j, t_0)\) in \( A \) is less than the amount of work done on \( \tau_j \) in \([A_j, t_0)\) in \( A' \), implies that \( \tau_j \) does not finish before \( t_0 \).

Let \( a \) be the maximum number in \([A_j, t_0)\) such that

\[
W(A, m(t), (2 - \frac{1}{m_{UB}}) \cdot s, \tau, a) \geq W(A', m(t), s, \tau, a)
\]

Notice that \( a < t_0 \). Such \( a \) must exist — \( a = r_j \) gives one.

We also know that \( a < t_0 \).

Consider two cases:

1. There is no \( k \) such that \( change_k > a \)
   
   Then let \( b = t_0 \).

2. There is a \( k \) such that \( change_k > a \)

   (a) There is no \( k \) such that \( change_k > a \) and \( change_k \leq t_0 \)

   Then let \( b = t_0 \).

   (b) There is no \( k \) such that \( change_k > a \) and \( change_k \leq t_0 \)

   Then let \( b = \min(change_k : change_k > a) \).

We will now study the time interval \([a, b)\), and let us summarize its properties:

\[
a < b
\]

\[
W(A, m(t), (2 - \frac{1}{m}) \cdot s, \tau, a) \geq W(A', m(t), s, \tau, a)
\]

\[
W(A, m(t), (2 - \frac{1}{m}) \cdot s, \tau, b) < W(A', m(t), s, \tau, b)
\]

\[
\forall t \in [a, b) \text{ it holds that } m(t) = m(a) \leq m_{UB}
\]

\( \tau \) has not finished at time \( b \) in the schedule generated by \( A \)

Let \( exec(t) \) denote the number of processors that were busy at time \( t \) in the case that tasks were scheduled by algorithm \( A \). Let \( x \) denote the cumulative length of time over the interval \([a, b)\) during which \( m(t) = exec(t) \). Let \( y = (b - a) - x \), that is, the length of time over the interval \([a, b)\) during which \( A \) idles some processor.

We make the following two observations:

- Since \( A \) is a work-conserving scheduling algorithm, \( \tau_j \), which has not finished by instant \( b \) in the schedule generated by \( A \), must have executed for at least \( y \) time units by time \( b \) in the schedule generated by \( A \); while it could have executed for at most \((x+y)\) time units in the schedule generated by \( A' \); therefore,

\[
(x + y) > (2 - \frac{1}{m_{UB}}) \cdot y
\]

- The amount of work done by \( A \) over \([a, b)\) is at least:

\[
(2 - \frac{1}{m_{UB}}) \cdot s \cdot (m(a) \cdot x + y)
\]

while the amount of work done by \( A' \) over this interval is at most

\[
m(a) \cdot (x + y) > (2 - \frac{1}{m_{UB}}) \cdot (m(a) \cdot x + y).
\]
By adding \( m(a) - 1 \) times Inequality 5 to Inequality 5, we get
\[
(m(a) - 1) \cdot (x + y) + m(a) \cdot (x + y) >
\]
\[
(m(a) - 1) \cdot \left(2 - \frac{1}{mUB}\right) \cdot y + (2 - \frac{1}{mUB}) \cdot (m(a) \cdot x + y)
\]
\[
\equiv (2m(a) - 1) \cdot (x + y) > (2 - \frac{1}{m(a)}) \cdot m(a) \cdot (x + y)
\]
\[
\Rightarrow (2m(a) - 1) \cdot (x + y) > (2 - \frac{1}{m(a)}) \cdot m(a) \cdot (x + y)
\]
which is a contradiction. □

We can now present a lemma that can be used to indirectly determine whether EDF meets deadlines.

**Lemma 3** Let \( A' \) denote an arbitrary scheduling algorithm. Let \( \pi \) denote a computer platform of \( m(t) \) processors and let \( \tau \) denote a computer platform that has, at every time, the same number of processors as \( \pi \), but the processors of \( \pi \) have speed \( 2 - \frac{1}{m_{edf}} \). If \( A' \) meets all deadlines of \( \tau \) on \( \pi \), then EDF meets deadlines of \( \tau \) on \( \pi \).

**Proof:** Since EDF is a work-conserving scheduling algorithm we obtain from Lemma 2 that for every \( t \):
\[
W(EDF, m(t), 2 - \frac{1}{m_{edf}}, \tau, s, t) \geq W(A', m(t), s, \tau, t)
\]

Let \( d_i = A_i + D_i \) and let \( \tau^k = \{\tau_1, \tau_2, \ldots, \tau_k\} \), where tasks are ordered in EDF priority, that is, \( d_1 \leq d_2 \leq \ldots d_k \).

We will prove the lemma using induction on \( k \).

**Base case** If \( k \leq m \) implies that a task is not delayed by another task and hence all deadlines hold.

**Induction step** We make two remarks. First, the scheduling of tasks \( \tau_1, \ldots, \tau_k \in \tau^{k+1} \) is the same as the scheduling of the tasks \( \tau_1, \ldots, \tau_k \in \tau^k \), so we only need to prove that \( \tau_{k+1} \) meets its deadline. Second, since \( \tau_{k+1} \) has the lowest priority according to EDF and there is no task with lower priority, \( \tau_{k+1} \), will do all work that the higher priority tasks does not do.

From the lemma we know that:

\( \text{all tasks in } \tau^{k+1} \text{ meet their deadlines when scheduled by } A' \text{ on } \pi \)

We can now reason as follows:

\( \text{all tasks in } \tau^{k+1} \text{ meet their deadlines when scheduled by } A' \text{ on } \pi \) ⇒

\[
W(A', m(t), 1, \tau^{k+1}, d_{k+1}) = \sum_{j=1}^{k+1} C_j \text{ use Lemma 2}
\]

\( W(EDF, m(t), 2 - \frac{1}{m_{edf}}, \tau^{k+1}, d_{k+1}) \geq \sum_{j=1}^{k+1} C_j \) ⇒

\( \tau_{k+1} \) executes at least \( C_{k+1} \) time units before \( d_{k+1} \) when scheduled by EDF on \( \pi \) ⇒

\( \tau_{k+1} \) meets its deadline when scheduled by EDF on \( \pi \) ⇒

\( \text{all tasks in } \tau^{k+1} \text{ meet their deadlines when scheduled by EDF on } \pi \)

The following lemma is a schedulability condition for EDF on a variable number of processors.

**Lemma 4** Consider EDF scheduling on \( m(t) \) processors. If \( \forall t: U(t) \leq m(t) \cdot \frac{mUB}{2mUB - 1} \) and \( C_i / D_i \leq \frac{mUB}{2mUB - 1} \) then all tasks meet their deadlines.

**Proof:** From the properties of OPT we know:

If a task set is such that \( \forall t: U(t) \leq m(t) \) and \( C_i / D_i \leq 1 \), then OPT meets all deadlines.

Applying Lemma 3 yields:

If a task set is such that \( \forall t: U(t) \leq m(t) \) and \( C_i / D_i \leq 1 \) and processors have the speed \( 2 - \frac{1}{m_{edf}} \), then EDF meets all deadlines.

Scaling the speed of processors yields:

If a task set is such that \( \forall t: U(t) \leq m(t) \cdot \frac{mUB}{(2mUB - 1)} \) and \( C_i / D_i \leq \frac{mUB}{(2mUB - 1)} \) and processors have the speed 1, then EDF meets all deadlines.

To be able to prove capacity bounds of task sets that have not only light tasks but also have heavy tasks, we introduce two new terms and present a lemma from previous research. Let \( heavy(t) \) denote the number of current tasks at time \( t \) that have \( C_i / D_i > \frac{m}{2mUB - 1} \) and let \( U_{light}(t) \) denote the sum of utilization of all current tasks at time \( t \) that have \( C_i / D_i \leq \frac{m}{2mUB - 1} \).

We will make use of a result by Ha and Liu [11], which states how the finishing time of a task is affected by the variability of execution times of tasks in global priority-driven scheduling. Let \( f_i \) denote the finishing time of the task \( \tau_i \),
Denote the finishing time of the task $\tau_i$ when all tasks execute at their maximum execution time and $f_{i^-}$ denote the finishing time of the task $\tau_i$ when all tasks execute at their minimum execution time. Lemma 5 presents the result that we will use.

**Lemma 5** For global scheduling where the priority orderings of tasks does not change when the execution times change, it holds that:

$$f_{i^-} \leq f_i \leq f_{i+}$$

**Proof:** See Corollary 3.1 in [11]. \qed

We can now design a schedulability condition for EDF-US$(m/(2m-1))$. Consider EDF-US$(m/(2m-1))$ scheduling on $m$ processors. If $\forall t: U_{light}(t) \leq (m - \text{heavy}(t)) \cdot m/(2m - 1)$ and $\forall t: \text{heavy}(t) \leq m - 1$, then all tasks meet their deadlines.

**Proof:** The tasks with $C_i/D_i > \frac{m}{2m-1}$ meet their deadlines because they receive the highest priority and there are at most $m-1$ of them. It remains to prove that tasks with $C_i/D_i \leq \frac{m}{2m-1}$ meet their deadlines. Consider two cases.

- All tasks with $C_i/D_i > \frac{m}{2m-1}$ have $C_i = D_i$. The tasks with $C_i/D_i \leq \frac{m}{2m-1}$ experiences it as if there were $m - \text{heavy}(t)$ processors available for them to execute on, and according Lemma 4 the tasks meet their deadlines.

- Of the tasks with $C_i/D_i > \frac{m}{2m-1}$, there is a subset of tasks that have $C_i < D_i$. If this subset of tasks had $C_i = D_i$, then according to the first case, all deadlines would hold. Reducing $C_i$ of tasks with $C_i/D_i > \frac{m}{2m-1}$ does not affect priority ordering so according to Lemma 5 all deadlines continue to hold. \qed

Now we have all the lemmas at our disposal for stating our final theorem.

**Theorem 1** Consider EDF-US$(m/(2m-1))$ scheduling on $m$ processors. If $\forall t: U(t) \leq m \cdot m/(2m - 1)$ then all tasks meet their deadlines.

**Proof:** It follows from $\forall t: U(t) \leq m \cdot m/(2m - 1)$ that: $\forall t: U_{light}(t) \leq (m - \text{heavy}(t)) \cdot m/(2m - 1)$ and $\forall t: \text{heavy}(t) \leq m - 1$. Applying Lemma 6 gives the theorem. \qed

Theorem 1 states that EDF-US$(m/(2m-1))$ has a capacity bound of $m/(2m - 1)$. For a large number of processors this bound approaches $1/2$. In Example 2 we show that an upper bound on the capacity bound of every priority-driven scheduling algorithm is $0.5 + 0.5/m$, which demonstrates that EDF-US$(m/(2m-1))$ is close to the best possible performance and with an infinite number of processors, no priority-driven scheduling algorithm can perform better than EDF-US$(m/(2m-1))$.

**Example 2** Consider $m + 1$ aperiodic tasks that should be scheduled on $m$ processors using priority-driven global scheduling. All tasks have $A_i = 0$, $D_i = 1$, $C_i = 0.5 + \epsilon$, so at every instant during $[0,1)$, the system utilization is $0.5 + \epsilon + \frac{0.5 + \epsilon}{m}$. Because of priority-driven scheduling, there must be a task with lowest priority, and that priority ordering is not permitted to change. That task executes when its higher priority tasks do not execute. Hence the lowest priority task executes $0.5 - \epsilon$ time units during $[0,1)$, but it needs to execute $0.5 + \epsilon$ time units, so it misses its deadline. We can do this reasoning for every $\epsilon > 0$ and for every $m$, so letting $\epsilon \to 0$ and $m \to \infty$ gives us that:

There are task sets that always have a system utilization arbitrarily close to $1/2 + 1/(2m)$, but no priority-driven scheduling algorithm can meet all its deadlines.

### 6 The design of a better admission control

In aperiodic online scheduling, we have no knowledge of future arrivals, which means that any sets of tasks could arrive. Some of those task sets could cause the system to become overloaded and in the worst case make all tasks miss their deadline. Hence, it is necessary to use an admission controller that can avoid such situations. A straightforward admission controller would be to only admit tasks that cause the resulting task set to satisfy the schedulability condition in Theorem 1. Unfortunately, such an approach has a serious drawback: Assume that $m$ tasks arrive and when they finish, all processors become idle. With our admission controller used so far, the utilization of these tasks must still be considered because their deadlines have not yet expired, which may lead to that no new tasks can be admitted. Clearly, this is an undesirable situation. However, we can design a better admission controller based on the following observation.

**Observation 1** For EDF-US$(m/(2m-1))$ the following holds: If all processors are idle at time $t$, then the schedule of tasks after time $t$ does not depend on the schedule before time $t$.

We can now design an improved admission controller, **Reset-all-idle**, which works as follows. A variable called **admission_counter** is initialized to zero when the system starts. When a task $\tau_i$ arrives, if $u_i/m$ plus admission_counter is no greater than the capacity bound,
then task \( \tau_i \) is admitted; otherwise, it is rejected. If \( \tau_i \) is admitted, then the admission counter is increased by \( u_i/m \). If all processors are idle then admission counter is reset to zero. When the deadline of a task expires, the admission counter is decreased by \( u_i/m \), if the task arrived after or at the time of the last reset. The goal of the new admission controller is to keep processors busy as much as possible while meeting deadlines of admitted tasks.

We will now evaluate the performance of this admission controller. To measure its performance, we define the real system utilization in the time interval \([t_1,t_2]\) as

\[
\frac{1}{m} \int_{t_1}^{t_2} \text{the number of busy processors at time } t \, dt.
\]

We expect that Reset-all-idle will keep the real system utilization higher than the capacity bound.

### 6.1 Experimental setup

Tasks are randomly generated with inter-arrival times between two subsequent tasks as exponentially distributed. Execution times and deadlines are generated from uniform distributions with the minimum value of 1. If the execution time of a task is greater than its deadline, then the execution time and the deadline is generated again. In all experiments, we choose the expected value of the deadline \( E[D] \) as 10,000, whereas different experiments use different expected value of the execution time \( E[C] \) and the number of processors. We generate tasks so that the first task arrives at time 0 and we generated new tasks until \( A_i + D_i > 10000000 \), then no more tasks were generated. When a task arrives, reset-all-idle decides whether it should be admitted and we schedule the admitted tasks during the time interval \([0,10000000]\). Hence, when we stopped the simulation, there were no tasks with remaining execution. We choose the expected value of the inter-arrival times so that the load, defined as:

\[
\frac{1}{m} \sum_{i=1}^{m} \frac{E[C_i]}{E[D_i]}
\]

is one. When we say real system utilization when we mean the real system utilization during \([0,10000000]\).

### 6.2 Experimental results

Figure 2 shows real system utilization as a function of the number of processors and \( E[C]/E[D] \). It can be seen for EDF-US\((m/(2m-1))\) that the fewer processors there are and the shorter the execution times of tasks are, the more the real system utilization exceeds the capacity bound. For example, for \( E[C]/E[D] = 0.02 \) and \( m = 3 \), the real utilization is 90%. In contrast, the capacity bound is 60%. The reason is that for these workloads, there are many instances when the admission controller can be reset. This happens when the execution times of tasks are small, because then the processors are more evenly utilized, and hence if one processor is idle, then it is more likely that all processors are idle. When there are few processors, the same explanation hold: if one processor is idle, then there is greater likelihood that all processors are idle.

### 7 Conclusion and future work

We have studied multiprocessor scheduling for aperiodic tasks where future arrivals are unknown. A previously proposed periodic priority-driven scheduling algorithm with task migration was extended to aperiodic scheduling and was shown to have a capacity bound of 0.5. This bound is close to the best a priority-driven scheduling algorithm can achieve. With an infinite number of processors, no priority-driven scheduling algorithm can perform better. We also proposed a simple admission controller which guarantees that admitted tasks met their deadlines and for many workloads, it admitted tasks so that the utilization could be kept above the capacity bound.

We have left open the problem of designing a priority-driven algorithm with a capacity bound such that no other priority-driven algorithm can have a greater capacity bound, even on a finite number of processors. In addition, it may be possible to find other conditions when one can safely reset the admission counter.
Figure 2: Real system utilization as a function of the number of processors and expected value of the utilization of tasks. The light shaded regions indicate where the real system utilization is greater than the capacity bound of the scheduling algorithm whereas the dark shaded regions indicate where the real system utilization is 50% greater than the capacity bound.

References


