Available online at www.sciencedirect.com
ScienceDirect
Journal of Computer and System Sciences 73 (2007) 225–241
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
www.elsevier.com/locate/jcss

Exact admission-control for integrated aperiodic and periodic tasks

Björn Andersson [*,1], Cecilia Ekelin [2]

Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

Received 1 July 2005; received in revised form 7 January 2006

Available online 27 October 2006

Abstract

Admission-controllers are used to prevent overload in systems with dynamically arriving tasks. Typically, these admission-controllers are based on sufficient (but not necessary) capacity bounds in order to maintain a low computational complexity. In this paper we present how *exact* admission-control for aperiodic tasks can be efficiently obtained. Our first result is an admission-controller for purely aperiodic task sets where the test has the same runtime complexity as utilization-based tests. Our second result is an extension of the previous controller for a baseload of periodic tasks. The runtime complexity of this test is lower than for any known exact admission-controller. In addition to presenting our main algorithm and evaluating its performance, we also discuss some general issues concerning admission-controllers and their implementation.

Keywords: Real-time systems; Schedulability analysis; Operating systems; Online scheduling; Earliest-deadline-first; AVL tree; Lazy evaluation

1. Introduction

In systems with dynamically arriving tasks, for example, web servers or real-time databases, it is typically not known when a task will arrive. If too many tasks arrive simultaneously, the system will become overloaded and tasks will miss their deadlines. The purpose of the admission-controller is to accept or reject arriving tasks based on the amount of available (remaining) capacity such that, once a task is accepted, it will be guaranteed to meet its deadline. The term capacity usually translates to utilization which is a well-defined concept for *periodic* tasks [1]. For *aperiodic* tasks, the definition of utilization is less intuitive [2] although previous work on admission-controllers use the so-called synthetic utilization [3]. The main difference between the utilization of periodic and aperiodic tasks is that for periodic tasks, the utilization represents the exact amount of requested capacity while for aperiodic tasks the capacity is overestimated. The reason is that the concept of utilization assumes that a task will never leave the system. This

* Corresponding author.
E-mail addresses: ba@ce.chalmers.se, bandersson@dei.isep.ipp.pt (B. Andersson), cekelin@ce.chalmers.se, cecilia.ekelin@volvo.com (C. Ekelin).
[1] The author is now with the IPP Hurray Research Group, Polytechnic Institute of Porto, Rua Dr. Antonio Bernardino de Almeida 431, 4200-072 Porto, Portugal.
[2] The author is now with the Department of Electronics and Software, Volvo Technology Corporation, Sven Hultins Gata 9C, SE-41288 Göteborg, Sweden.

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2006.04.001

implies that capacity bounds used in admission-controllers for aperiodic tasks are unnecessarily pessimistic. This pessimism is increased further by the fact that admission-controllers tend to offer only a sufficient admission test [4]. The reason for this is that it is generally believed that an exact admission-controller would have a too high computational complexity. Since running the admission-control algorithm takes time from running the tasks, it is crucial that its run-time complexity be low. Admission-controllers based on capacity bounds are claimed to run in constant time ($O(1)$) [3]. These admission-controllers need to decrement a counter when the deadline of a task expires. In the restricted case when tasks can be partitioned into service classes, such that all tasks in a service class has the same relative deadline, and the number of service classes is bounded then the computational complexity is indeed $O(1)$. But in the general case the complexity is $O(\log n)$ (where n is the number of tasks) [5] because a data structure of accepted tasks must be maintained so that the task with the minimum absolute deadline can be retrieved. Clearly, it is desirable to design *exact* $O(\log n)$ admission tests since the real processor utilization would then be higher compared to sufficient tests while the overall complexity remains unchanged. In this paper, we propose such an exact admission-controller for aperiodic tasks.

So far we have assumed that all tasks are aperiodic. However, many real-time systems consist of a baseload of hard periodic tasks where aperiodic tasks may be executed as long as they do not cause any periodic task to miss its deadline. Although the m periodic tasks can be treated as a number of aperiodic ones—by unrolling their executions within some time frame p—this typically results in a runtime complexity of $O(p)$ where $p \gg m$. Instead, we propose an exact admission-controller with complexity $O(m + \log p + \log n)$. In addition to performing exact admission-control, our approach also exhibits a number of interesting features:

(1) In the extreme case, the cumulative value (total execution-time of accepted tasks) of our admission-controller approaches infinity while it approaches zero for utilization-based controllers.
(2) The run-time complexity of our admission-controller becomes smaller as the load in the system increases.
(3) If a task executes for a time shorter than its declared execution-time, the unused capacity is available again for the admission-controller.
(4) If a task exceeds its given execution-time, the scheduler has information to decide whether this will affect the other tasks or not.

By generating tasks randomly and scheduling them by our exact admission-controller and the utilization-based controller proposed in [6] we find that the utilization-based controller indeed is wasteful compared to our exact admission-controller.

The rest of this paper is organized as follows. Section 2 defines the system model. Section 3 describes our admission-control algorithm (for purely aperiodic task sets) and in particular the data structure it uses. Section 4 discusses some properties of that admission-controller. Section 5 contains our experimental results. In Section 6 we show how our admission-control algorithm can be extended to handle a baseload of periodic tasks. In Section 7 we discuss related and future work while Section 8 summarizes our current work.

2. System model

We will begin by considering the problem of scheduling a task set $\mathcal{T}$ of n aperiodically-arriving real-time tasks on a single processor. An aperiodic task τ_i has an arrival time A_i, an execution-time C_i and a relative deadline D_i, that is, the task requests to execute C_i time units during the time interval $[A_i, A_i + D_i)$. For convenience, we define the absolute deadline d_i as $d_i = A_i + D_i$. We assume that C_i and D_i are positive real numbers such that $C_i \leqslant D_i$ and A_i is a real number. Without loss of generality we will assume $0 \leqslant A_1 \leqslant A_2 \leqslant \cdots \leqslant A_n$.

We will then extend the problem to also include a task set $\mathcal{T}_{\text{per}}$ consisting of m periodically-arriving real-time tasks that are known a priori. A periodic task τ_i is characterized by a release time R_i, an execution-time C_i, a period T_i and a deadline D_i. An *invocation* τ_i^k of a periodic task is supposed to execute once within the interval $[R_i + T_i \cdot (k-1), R_i + T_i \cdot (k-1) + D_i]$. We will assume that $R_i = 0$ and $D_i = T_i$. Thus, $A_i^k = T_i \cdot (k-1)$ and $d_i^k = A_i^k + T_i$. Moreover, we assume that $U_{\text{per}} = \sum_{i=1}^{m} \frac{C_i}{T_i} \leqslant 1$.

We study EDF (earliest-deadline-first) [1] scheduling which (without admission-controller) behaves as follows. Tasks that have arrived and are awaiting execution are kept in a queue, called *ready queue*, sorted ascendingly by their absolute deadlines. When the processor becomes idle, the first task in the queue is selected for execution. When a task

arrives it is inserted in the queue (breaking ties arbitrarily). If the deadline of this newly arrived task is shorter than the deadline of the currently running task, the latter is preempted and the new task starts to execute instead. We assume that a task always can be preempted and there is no cost of preemption.

An admission-controller acts as a filter for arriving tasks such that a task is only allowed into the system if it is guaranteed that all tasks in the ready queue *and* future arrivals of periodic tasks will still meet their deadlines with the given scheduling algorithm. It is assumed that the admission-controller (or scheduling algorithm) is not allowed to use information about future aperiodic tasks, that is, at time t it is not allowed to use A_i, D_i or C_i of $\tau_i \in \mathcal{T}$ with $A_i > t$. Thus, the admission-control problem is as follows:

> Given the task set $\mathcal{T}_{\text{per}}$ of periodic tasks and the task set $\mathcal{T}_{\text{aper}}$ of previously admitted aperiodic tasks, can aperiodic task τ_i be admitted?

In Section 3 we will propose an admission-controller under the assumption that $\mathcal{T}_{\text{per}} = \emptyset$ while an algorithm for the general case will be proposed in Section 6.

3. The admission-controller

Instead of using an aggregate of the task properties, such as the utilization, we base our admission-controller on the actual properties which contain more information and thus enable an exact analysis. The drawback is that data structures used to maintain these properties become harder to implement and possibly time-consuming to update. Therefore, when devising an exact admission-controller it is not only the idea of the algorithm that is important but also how to implement it efficiently. In the description of our admission-controller we will treat these two aspects separately to simplify the understanding. Furthermore, as a start, we will assume that $0 = A_1 = A_2 = \cdots = A_n$ (which implies that $D_i = d_i$) and that no d_i are the same.

3.1. Algorithm description

It is known [7], that if and only if the following condition holds, all tasks (in the ready queue) will meet their deadlines:

- *Schedulability condition:* $\forall \tau_i \colon \sum_{d_j \leqslant d_i} C_j \leqslant d_i$.

The basic idea of our algorithm is to use this condition as the admission test. That is, if an arriving task would cause the condition to be violated, the task is rejected, otherwise it is accepted. Unfortunately, to check whether the condition is satisfied, all tasks in the queue may have to be traversed, resulting in a run-time complexity of $O(n)$. However, it is possible to make the test in $O(\log n)$ by introducing some additional task parameters. We will use the notation $\tau_{\text{pos}(k)}$ to denote the task at position k in the queue. Thus, $\tau_{\text{pos}(k-1)}$ and $\tau_{\text{pos}(k+1)}$ indicate the immediately preceding and succeeding tasks, respectively. Note that, for $l < k$, $d_{\text{pos}(l)} < d_{\text{pos}(k)}$ and for $l > k$, $d_{\text{pos}(l)} > d_{\text{pos}(k)}$. For each task τ_i at position k in the queue we define the following two task parameters:

- The accumulated execution-time of preceding tasks,

$$e_i = \sum_{l=1}^{k-1} C_{\text{pos}(l)}.$$

- The minimum slack of succeeding tasks,[3]

$$s_i = \min\{d_{\text{pos}(l)} - e_{\text{pos}(l)} - C_{\text{pos}(l)} \colon \forall l > k\}.$$

An illustration of these parameters can be found in Fig. 1. Note that $e_{\text{pos}(1)} = 0$ and $s_{\text{pos}(n)} = \infty$. The admission

[3] It is assumed that $\min\{\} = \infty$.

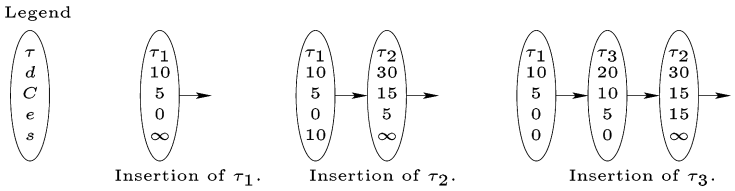


Fig. 1. Illustration of the ready queue and resulting schedule.

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
C_i	5	15	10	5	50	10	1	2	1	1
D_i	10	30	20	50	100	40	80	60	45	65

Fig. 2. Example 1 task set.

test then works as follows. When a new task τ_i arrives, its potential position[4] k in the queue is looked up, that is, the information for tasks $\tau_{\text{pos}(k+1)}$ and $\tau_{\text{pos}(k-1)}$ is collected.[5] This lookup procedure can be done in $O(\log n)$ if the queue is implemented as a balanced tree with d_i as keys. The task is then accepted if the following conditions are met:

Condition 1. The new task τ_i will meet its deadline. $e_{\text{pos}(k+1)} + C_i \leqslant d_i$.

Condition 2. The succeeding tasks will continue to meet their deadlines. $s_{\text{pos}(k-1)} \geqslant C_i$.

Note that the schedulability of preceding tasks cannot be affected. Also note that these two conditions are equivalent to the schedulability condition implying that the admission test still is exact. If the admission test succeeds, the task is inserted in the already retrieved position with $e_i = e_{\text{pos}(k+1)}$ and $s_i = s_{\text{pos}(k-1)} - C_i$. This also results in that the information for the other tasks in the queue must be updated. For all succeeding tasks ($l > k$) $e_{\text{pos}(l)} := e_{\text{pos}(l)} + C_i$ and $s_{\text{pos}(l)} := s_{\text{pos}(l)} - C_i$ while for all preceding tasks ($l < k$) $s_{\text{pos}(l)} := \min\{s_{\text{pos}(l)}, d_i - e_i - C_i, s_i\}$.

Clearly, although the admission test now can be done in $O(\log n)$, the overall run-time complexity is still $O(n)$ since on accept we may have to update all tasks in the queue. However, as we will show in the next section, we can use an AVL-tree and a form of lazy evaluation to obtain an $O(\log n)$ algorithm. It is important to note that the overall $O(\log n)$ complexity cannot be achieved simply by using an AVL-tree since the tree by itself only provides $O(\log n)$ complexity for find/insert/delete operations concerning a *single* entry. In our case we may have to update all n entries. Thus a major contribution of this paper is showing how these n updates can be performed in $O(\log n)$.

Example 1. Consider the task set in Fig. 2. When τ_1 arrives (to the admission-controller) the queue is empty which means that the admission test is $C_1 \leqslant d_1$, i.e., $5 \leqslant 10$ which holds. Hence, the task is accepted and inserted with $e_1 = 0, s_1 = \infty$. τ_2 should be after τ_1 since $d_2 > d_1$ so the admission test becomes $e_1 + C_1 + C_2 \leqslant d_2$, i.e., $0 + 5 + 15 \leqslant 30$ which holds and τ_2 is inserted with $e_2 = 0 + 5 = 5, s_2 = \infty - 5 = \infty$. τ_1 is then updated with $s_1 := \min\{\infty, 30 - 5 - 15, \infty\} = 10$. τ_3 is to be inserted between τ_1 and τ_2 implying that both Conditions 1 and 2 must be met, that is, $5 + 10 \leqslant 20$ and $10 \geqslant 10$ which holds so the task is accepted with $e_3 = 5, s_3 = 10 - 10 = 0$. τ_2 is updated with $e_2 := 5 + 10 = 15$ ($s_2 := \infty - 10 = \infty$) and for τ_1, $s_1 := \min\{10, 20 - 5 - 10, 0\} = 0$. The next task, τ_4, is to be last, yielding the test $15 + 15 + 5 \leqslant 50$ which holds and results in $e_4 = 15 + 15 = 30, s_4 = \infty$. τ_2 is updated as $s_2 := \min\{\infty, 50 - 30 - 5, \infty\} = 15$ whereas for τ_3 we see that $s_3 (= 0) < 50 - 30 - 5$ which means that no more preceding tasks (τ_1) need to be updated.[6]

The insertion of τ_1, τ_2 and τ_3 and the corresponding schedule are illustrated in Fig. 1. In the end, all ten tasks will be accepted and have a minimum slack of zero (except τ_5). This tells us that the utilization is 100% and that no task with $d_i \leqslant 100$ can be accepted. This performance can be compared with a utilization-based admission-controller which would have rejected all tasks but τ_1 and τ_2 since their combined synthetic-utilization is $\frac{5}{10} + \frac{15}{30} = 1$. (The admission test is $U_{\text{synthetic}} = \sum_{i=1}^{n} \frac{C_i}{D_i} \leqslant 1$.) Thus, the real utilization would only be $\frac{5+15}{100} = 0.2$.

[4] This means the position after a speculative insert.

[5] If there is no succeeding/preceding task, data for the preceding/succeeding task is used instead. That is, $e_{\text{pos}(k+1)} = e_{\text{pos}(k-1)} + C_{\text{pos}(k-1)}$ and $s_{\text{pos}(k-1)} = \min\{s_{\text{pos}(k+1)}, d_{\text{pos}(k+1)} - e_{\text{pos}(k+1)} - C_{\text{pos}(k+1)}\}$.

[6] The reason is that $s_{\text{pos}(l)} \leqslant s_{\text{pos}(k)}$ holds for $l < k$.

3.2. Data structure

We use an AVL-tree [8] as the basic data-structure for the ready queue (although our idea is likely to work for any balanced tree-structure). An AVL-tree is a binary tree, ordered such that for an entry E_i, entries E_j with $\text{key}(E_j) < \text{key}(E_i)$ ($\text{key}(E_j) > \text{key}(E_i)$) are found in the left (right) subtree of E_i. Each entry records the balance as $-$, $+$ or 0 representing a skew to the left, right or none in the height of its subtrees. If an insert or delete operation results in a skew of more than one, rotations of subtrees are performed to reestablish the balance. An AVL-tree guarantees that operations such as find, insert and delete are done in $O(\log n)$ steps. Now, what we want to do is to also update the e and s values for all n entries in the tree in $O(\log n)$ steps. This may sound impossible but is actually achievable in our case since (i) an update is the same for all preceding/succeeding entries (ii) an entry need only be fully up-to-date when requested. This works as follows.

When a task arrives, the admission-controller will look up its potential position in the AVL-tree (which is ordered by the task deadlines). This is done by starting at the root entry and traversing the tree by selecting either the left or right subtrees. This means that there will be a path from the root to the leaf containing those entries that are traversed. If the admission-controller then accepts the task, its entry is simply added at the end of the path. However, we must now make the necessary updates of the e and s values for succeeding and preceding tasks. Due to the constitution of the tree, we know that, for all entries in the path, the deadlines in the left (right) subtrees are shorter (longer) than for the new task. Hence, a subtree either contains only preceding or succeeding tasks. (In contrast, the path may contain both preceding and succeeding tasks.) This means that it is enough to update the values for the entries on the path, since when doing so we can make a *note* for each entry saying that the next time the left (or right) subtree is traversed, the e and s values should be set/increased/decreased by a certain amount. The next time the admission-controller uses the look-up procedure the latter will perform the updates for those entries that it traverses. This includes moving the notes to the subtrees. The updates that can occur are as mentioned (i) $s_j := \min\{d_i - e_i - C_i, s_i\}$ (for tasks preceding the new task τ_i) and (ii) $s_j := s_j - C_i, e_j := e_j + C_i$ (for tasks succeeding τ_i). Hence, an update note contains two entities:

- The new minimum slack of succeeding tasks,

$$s_{\text{set}} = \{ -1, \text{if only relative change}, \geqslant 0, \text{if absolute (and possibly relative) change.}$$

- The increase in accumulated execution-time, $e_{\text{inc}} \geqslant 0$.

When the task τ_i is inserted the note for *succeeding* tasks is $s_{\text{set}} = -1, e_{\text{inc}} = C_i$ since the new task pushes the tasks backwards. Thus, the accumulated execution-time increase while the minimum slack decrease uniformly for all succeeding tasks. For *preceding* tasks the note is $s_{\text{set}} = \min\{d_i - e_i - C_i, s_i\}, e_{\text{inc}} = 0$ since the new task may cause a change in the minimum slack depending on the previous values but the accumulated execution-time is unaffected. When a note is to be posted or moved to the subtrees it may be the case that the subtree already has a previous note. We must then merge the information in the notes. For the accumulated execution-time the values should always be added since this value always increase, that is, $e_{\text{inc}}^{\text{new}} := e_{\text{inc}} + e_{\text{inc}}^{\text{old}}$. For the slack information we have four different cases:

Case 1. $s_{\text{set}} < 0$ and $s_{\text{set}}^{\text{old}} < 0$. No change, i.e., $s_{\text{set}}^{\text{new}} := s_{\text{set}}^{\text{old}}$.

Case 2. $s_{\text{set}} < 0$ and $s_{\text{set}}^{\text{old}} \geqslant 0$. All values have been uniformly affected including the slack information in the old note. Thus, we update the note as $s_{\text{set}}^{\text{new}} := s_{\text{set}}^{\text{old}} - e_{\text{inc}}$.

Case 3. $s_{\text{set}} \geqslant 0$ and $s_{\text{set}}^{\text{old}} < 0$. New information on the minimum slack, i.e., $s_{\text{set}}^{\text{new}} := s_{\text{set}}$.

Case 4. $s_{\text{set}} \geqslant 0$ and $s_{\text{set}}^{\text{old}} \geqslant 0$. The least of the minimum slacks should be used. Note that the change in the accumulated execution-time affects the old slack information. That is, $s_{\text{set}}^{\text{new}} := \min\{s_{\text{set}}^{\text{old}} - e_{\text{inc}}, s_{\text{set}}\}$.

When moving a note we must also remember not to update the slack for an entry if its minimum slack already is less than stated in the note since this indicates that the note is obsolete. That is, $s_j := \min\{s_j - e_{\text{inc}}, s_{\text{set}}\}$ for $s_{\text{set}} \geqslant 0$.

This data structure implies that an entry is only updated when it is requested which is in fact the only time its information is required to be correct. Hence, the computational complexity of keeping the tree up-to-date is indeed $O(\log n)$.

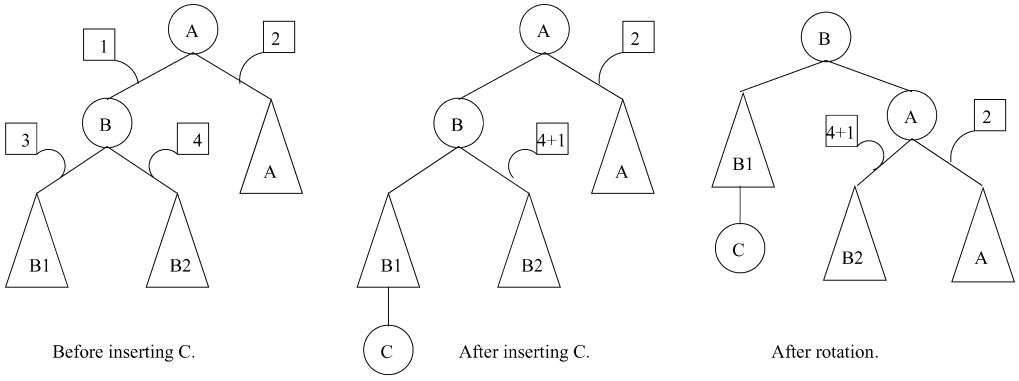


Fig. 3. Illustration of how the update notes are affected by the AVL-rotations.

It should be mentioned that when a new entry is inserted (or deleted) the possible rotations in the tree will cause subtrees to be moved. That is, left subtrees may become right subtrees and vice versa. However, this does not affect the validity of the update notes since these concern the relationship between *tasks* which is independent of the tree representation. (When the rotations are performed the notes also move along.) Furthermore, since seeing an entry means updating it, the actual rotations will only be performed on entries that are up-to-date which guarantees that consistency among notes is maintained.

As a proof sketch we show one case of the rotations in Fig. 3. In the figure the tree is first shown before the insertion of an entry C. As can be seen there are several update notes present. However, as C is inserted the update notes along its path will be carried out such that after the insertion all entries on the path will be completely updated. That is, entry B is updated with note 1 and the entries on the path in subtree B1 are updated with the merged notes of 1 and 3. Furthermore, subtrees that are not traversed receive new or merged update notes. For example, note 4 is merged with note 1. As can then be seen, the rotation only move subtrees between entries on the path. Since these entries and subtrees are bound to be updated due to the insert operation, the rotation does not cause inconsistency in the update notes. The same reasoning can be applied and validated for all cases of rotations that can occur during insert and delete.

The pseudo-code for our admission-controller can be seen in Fig. 4.

Example 2. We will use the task set in Fig. 2 again but this time we will also consider that the ready queue is implemented as an AVL-tree. After inserting $\tau_1 - \tau_5$ the tree will look like in Fig. 5(a). τ_6 should now be inserted to the right of τ_2. We get $e_6 = 30, s_6 = 15 - 10 = 5$ and have to update the tasks on the path, namely τ_3, τ_4 and τ_2. For τ_3 we have $s_3 := \min\{0, 40 - 30 - 10, 5\} = 0$ and since $0 \leqslant 40 - 30 - 10$ we do not have to update any entries left of τ_3 (τ_1). For τ_4 we have $s_4 := 15 - 10 = 5, e_4 := 30 + 10 = 40$ and for the right subtree we add the note $s_{\text{set}} = -1, e_{\text{inc}} = 10$. For τ_2 we have $s_2 := \min\{15, 40 - 30 - 10, 5\} = 0$. The resulting tree can be seen in Fig. 5(b) including an AVL-rotation to keep the tree balanced. When the position for τ_7 is looked up (left of τ_5), the note on the path is moved along to the entry for τ_5 which then is updated as $e_5 := 35 + 10 = 45$ ($s_5 := \infty - 10 = \infty$). τ_7 is inserted with $e_7 = 45, s_7 = 5 - 1 = 4$. The tasks τ_2, τ_4 and τ_5 on the path are updated. Again, as $s_2 < \min\{d_7 - e_7 - C_7, s_7\}$ no update (note) is necessary for tasks left of τ_2. However, for τ_4 we have $s_4 := \min\{5, 80 - 45 - 1, 4\} = 4$ which means that the left subtree needs a note. That is, $s_{\text{set}} = 4, e_{\text{inc}} = 0$. τ_5 is updated as $e_5 := 45 + 1 = 46$ ($s_5 := \infty - 1 = \infty$). This is all illustrated in Fig. 5(c). When τ_8 is inserted the updates are similar to those for τ_7. The difference is that now a note already exist and have to be updated. In this case, (a Case 4 update) the slack information should be replaced. When τ_9 is inserted the update is performed and instead a new note is added to the right of τ_4. This tree is shown in Fig. 6(a). When τ_{10} is to be inserted (to the right of τ_8) the note will be pushed down to the right of τ_7 while the entry for τ_8 is updated. The insertion will also cause the note to be updated and since it is a Case 1 update, the slack information will not change. We also get a new note to the left of τ_4. However, as the tree becomes unbalanced we have to do an AVL-rotation which will make this left note a right note as can be seen in Fig. 6(b). As mentioned, this will not affect the validity of the note since its information applies to all entries in a particular subtree regardless of the position of the subtree.

PROCEDURE: **admission-control**
Input: AVL-tree implementation of the ready queue and the parameters for τ_i
Output: Accept/Reject (and a possibly modified ready queue)

(1) Lookup the potential position k for τ_i in the tree (ordered by deadlines) and store the path. At the same time, perform postponed updates and move update notes to subtrees.
FOR all tasks τ_j on the path DO
  IF the link to τ_j contains an update note THEN
    $e_j := e_j + e_{\text{inc}}$
    IF $s_{\text{set}} \geqslant 0$ AND $s_j - e_{\text{inc}} > s_{\text{set}}$ THEN
      $s_j := s_{\text{set}}$
    ELSE
      $s_j := s_j - e_{\text{inc}}$
    END IF
    Move and merge update notes to left and right subtrees of τ_j
  END IF
END FOR

(2) Perform the admission test.

IF $e_{\text{pos}(k+1)} + C_i \leqslant d_i$ AND $s_{\text{pos}(k-1)} \geqslant C_i$ THEN
  Accept
ELSE
  Reject
END IF

(3) Insert the task, update path entries and create update notes.
IF Accept THEN
  Insert τ_i with $e_i = e_{\text{pos}(k+1)}, s_i = s_{\text{pos}(k-1)} - C_i$
  FOR all tasks τ_j on the path DO
    IF $d_j < d_i$ THEN
      IF $s_j > \min\{d_i - e_i - C_i, s_i\}$ THEN
        Merge note on left subtree with the note
        $s_{\text{set}} = \min\{d_i - e_i - C_i, s_i\}, e_{\text{inc}} = 0$
      END IF
      $s_j := \min\{s_j, d_i - e_i - C_i, s_i\}$
    ELSE
      $e_j := e_j + C_i, s_j := s_j - C_i$
      Merge note on right subtree with the note $s_{\text{set}} = -1, e_{\text{inc}} = C_i$
    END IF
  END FOR
END IF

(4) Perform any AVL-rotations to keep the tree balanced.

Merging of update notes
IF no previous update note exists THEN
  $e_{\text{inc}}^{\text{old}} := 0, s_{\text{set}}^{\text{old}} := -1$
END IF
$e_{\text{inc}}^{\text{new}} := e_{\text{inc}}^{\text{old}} + e_{\text{inc}}$
IF $s_{\text{set}} < 0$ THEN
  IF $s_{\text{set}}^{\text{old}} < 0$ THEN
    $s_{\text{set}}^{\text{new}} := s_{\text{set}}^{\text{old}}$
  ELSE
    $s_{\text{set}}^{\text{new}} := s_{\text{set}}^{\text{old}} - e_{\text{inc}}$
  END IF
ELSE
  IF $s_{\text{set}}^{\text{old}} < 0$ OR $s_{\text{set}} < s_{\text{set}}^{\text{old}} - e_{\text{inc}}$ THEN
    $s_{\text{set}}^{\text{new}} := s_{\text{set}}$
  ELSE
    $s_{\text{set}}^{\text{new}} := s_{\text{set}}^{\text{old}} - e_{\text{inc}}$
  END IF
END IF

Fig. 4. Pseudo-code for our admission-controller.

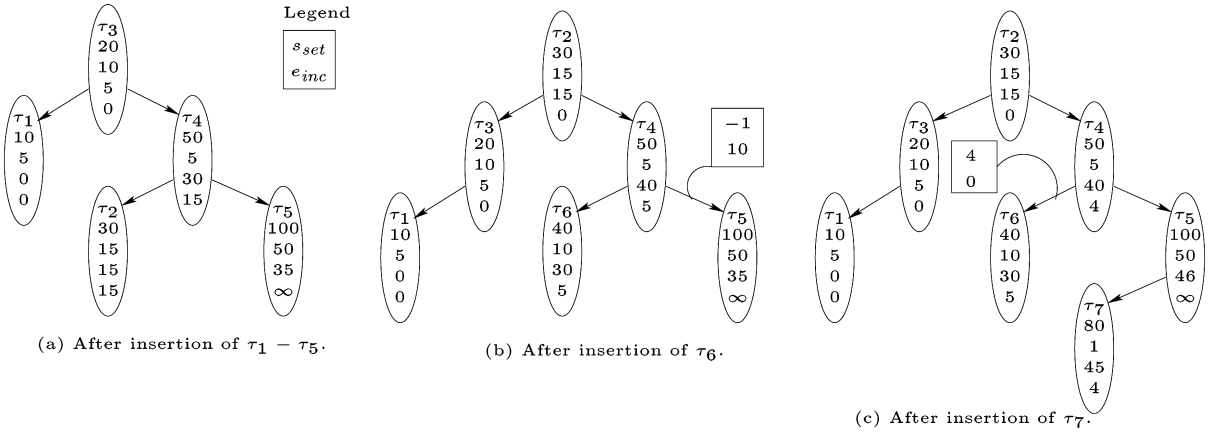


Fig. 5. Illustration of the AVL-tree.

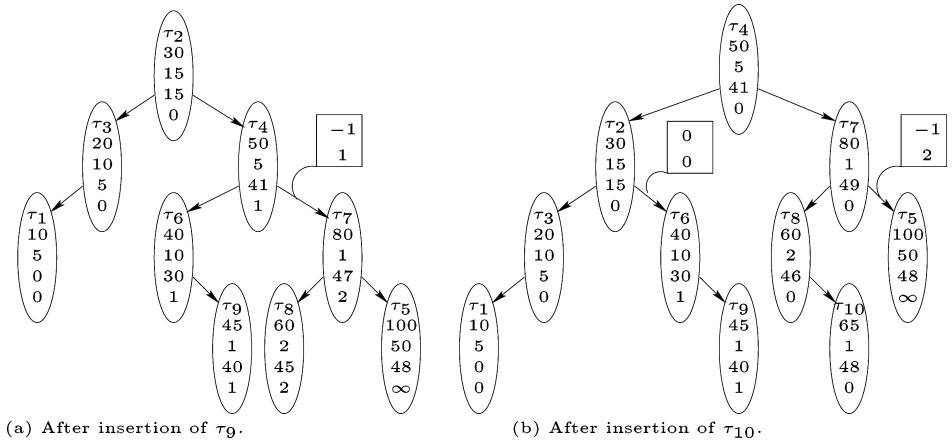


Fig. 6. Illustration of the AVL-tree.

3.3. Scheduler interaction

We will now remove the assumption that $0 = A_1 = A_2 = \cdots = A_n$, that is, we will consider how the ready queue (AVL-tree) is updated due to interaction with the EDF scheduler over time. First of all, the parameter e no longer represents the accumulated execution-time of preceding tasks but rather the scheduled start-time. For instance, when the ready queue is empty, $e_{\text{pos}(1)} = A_1$.

When the processor is idle and there are tasks in the ready queue, the scheduler selects the first task $\tau_{\text{pos}(1)}$ for execution. However, the task is not removed from the queue until it is finished. Instead, the scheduler keeps a pointer to its entry in the AVL-tree. When a new task τ_i arrives, the entry for $\tau_{\text{pos}(1)}$ is updated to indicate what portion of its code it has executed. That is, $C_{\text{pos}(1)} := C_{\text{pos}(1)} - c_{\text{pos}(1)}$ and $e_{\text{pos}(1)} := e_{\text{pos}(1)} + c_{\text{pos}(1)}$ where $c_{\text{pos}(1)}$ is the time the task has executed from its (last) dispatch. If τ_i is accepted, the scheduler has to redo the selection of which task to run. This selection also has to be done whenever a task is finished (even if there are preempted tasks). The removal of a task, when finished, requires at most $O(\log n)$ AVL-rotations to keep the tree balanced. Removal also includes performing any update notes encountered during the operation.

Non-unique d_i can be quite easily handled by being consistent in the deadline comparisons and keeping track of whether the entries on the path are to the left or right of their parent. That is, this is no different from an ordinary AVL-tree where entries may have similar keys.

Example 3. Consider a ready queue that contains two tasks τ_1 and τ_2 as in Fig. 1. If at $t = 2$ a new task arrives, the scheduler updates the entry for τ_1 with $C_1 := 5 - 2 = 3$ and $e_1 := 0 + 2 = 2$ before the admission-controller is

invoked. If the new task is accepted and has a shorter deadline than τ_1 (which is currently running), the new task will be the one to run and it will preempt τ_1. However, due to the previous update, this preemption is transparent to the admission-controller since the amount of requested capacity still is accurate. (Of course, the management of the actual context-switch is handled by the processor/scheduler.)

4. Algorithm properties

Here we will discuss some properties of our admission-controller and its implementation.

4.1. An infinite improvement

As mentioned, utilization-based admission-controllers suffer from overestimation of the required capacity. This means that the *cumulative value* Γ for such admission-controllers typically is much lower than for ours. The cumulative value is (in our case) defined as $\Gamma = \sum_{i=1}^{n} b_i \cdot C_i$ where $b_i = 1$ if τ_i is accepted and zero otherwise. It can happen that, $\Gamma_{\text{exact}} = \infty$ while $\Gamma_{\text{util}} = \epsilon$ where ϵ is arbitrarily close to zero. This is shown in the following example.

Consider two tasks τ_1 and τ_2 where $0 < C_1 < D_1$ and the execution-time of τ_2 is $C_2 = 2 \cdot D_1 - C_1$ and its deadline $D_2 = 2 \cdot D_1$. Let $A_1 = A_2 = 0$. Clearly, τ_1 can be accepted since $C_1 < D_1$ and there are no tasks in the ready queue. Since $\frac{C_1}{D_1} < 1$ an admission-controller based on utilization bounds will also accept the task. For τ_2 the test $e_1 + C_1 + C_2 \leqslant D_2$ succeeds since $0 + C_1 + 2 \cdot D_1 - C_1 \leqslant 2 \cdot D_1$ holds. However, the utilization is $\frac{C_1}{D_1} + \frac{2 \cdot D_1 - C_1}{2 \cdot D_1} = \frac{2 \cdot D_1 + C_1}{2 \cdot D_1} > 1$ which means that τ_2 will be rejected. If $D_1 \to \infty$ and $C_1 \to 0$, $\Gamma_{\text{exact}} \to \infty$ while $\Gamma_{\text{util}} \to 0$.

4.2. Overhead becomes lower as load increases

In Condition 2 in the admission test, it can be seen that if $s_{\text{pos}(k+1)} < C_i$ the task cannot be accepted. In fact, since $s_{\text{pos}(k+1)} \leqslant s_{\text{pos}(k+x)}$ (where $x > 1$) we know that, as soon as we have seen an entry with task τ_j where $d_j > d_i$ and $s_j < C_i$ the task τ_i will have to be rejected. This is particularly useful if the system is heavily loaded since the admission-controller will be able to reject tasks without traversing the entire path. Thus, as the load increases the number of entries on the path decreases until ultimately only the root entry needs to be checked. (This reasoning assumes that deadlines of arriving tasks follow the same distribution as the tasks in the ready queue.) It is also in severe overload situations that the run-time overhead of the admission-controller matters the most since a lot of time will have to be spent on admission decisions.

4.3. QoS negotiation

In quality-of-service negotiations a task interacts with the admission-controller/scheduler to establish how much capacity it may request and in what time this capacity can be delivered. That is, the task suggests an execution-time and deadline but may be willing to relax these demands if it cannot be scheduled. In utilization-based admission-controllers this kind of negotiation is simply done by executing a schedulability test on the suggested parameters and then report success or failure. Hence, it is the responsibility of the task to change the parameters as it sees fit. Since the task does not know anything about the tasks in the ready queue, it would be more effective if the admission-controller could suggest suitable modifications. In our case, this is possible due to the information maintained for each task. For example, if the task only suggests a deadline, the maximum allowed execution-time for the task simply is $s_{\text{pos}(k-1)}$ which is obtained from the look-up procedure. Moreover, if an execution-time is suggested, the shortest allowed deadline can be returned. This is done by traversing the tree, turning left (right) whenever $s_j \geqslant C_i$ ($s_j < C_i$). The deadline will then be $e_j - A_i + C_i$ where E_j is the stopping entry. Although these suggestions can be made also with an utilization-based admission-controller, ours also has the ability to produce a list of schedulable execution-time, deadline combinations. That is, for each task in the ready queue, s_j equals the maximum C_i of a new task with $D_i = d_j - e_j$. The gathering of such a list requires $O(n)$ steps but may be useful for tasks with flexible demands.

4.4. Optimal admission-control

To compare the performance of on-line scheduling algorithms the concept of competitive factor φ is used [9]. This parameter is given by the relationship $\Gamma \geqslant \varphi \cdot \Gamma_{\text{opt}}$ where $0 \leqslant \varphi \leqslant 1$. It is usually assumed that Γ_{opt} is the cumulative

value for an clairvoyant scheduler. That is, if the scheduler/admission-controller had knowledge about the future, it would have the freedom to reject a task although the admission test succeeds, in order to accommodate "better" future arrivals. It is always possible to construct a task set such that it is beneficial to reject schedulable tasks implying that no optimal admission-controller can exist for this definition of optimality. However, clairvoyance is not available in reality, making this definition useless for performance comparison since most admission-controllers will have $\varphi = 0$. Instead, it is more reasonable to assume that, if a task *can* be scheduled, the admission-controller *must* accept the task. With this additional requirement, our admission-controller has $\varphi = 1$ since it performs exact schedulability analysis. Thus, our admission-controller is optimal.

4.5. Capacity overestimation

Our system model assumes that a task executes exactly C_i time units. However, in reality, the execution-time of a task typically varies with the input and the state of the processor. Therefore, C_i often represents the worst-case execution-time of a task in order to make the schedulability analysis safe. Unfortunately, this means that the average execution-time of a task typically is much shorter than assumed by the admission-controller which causes tasks to be rejected although there is enough capacity. In our case, the unused capacity can be made available again by keeping track of the spare capacity. If the task only executed for $c < C_{\text{pos}(1)}$ time units, this means that the scheduled start-times have been moved forward and the minimum slack has increased. Thus, for all tasks $e_j := e_j - C_{\text{pos}(1)} + c$ and $s_j := s_j + C_{\text{pos}(1)} - c$. This update can be done in $O(1)$ by using a global variable $x := x + c - C_{\text{pos}(1)}$ that is introduced in the tests. That is, e and s are replaced by $e + x$ and $s - x$. (When the processor becomes idle x is reset to zero.)

4.6. Overrun protection

Aperiodic tasks are often handled by using so-called servers such as the constant-bandwidth server [10]. The advantage of such servers over utilization-based admission-control is claimed to be task isolation. That is, if a task executes for longer than anticipated, it will be terminated (or postponed) and the schedulability of the other tasks will not be affected. This is the case since aperiodic tasks are only allowed to use a certain total amount of execution-time that is dictated by the server budget. Thus, when the capacity of the budget is ended so are the tasks. The same kind of protection is not available in ordinary EDF where a task is executed until it is finished. This means that an overrun may cause all other accepted tasks to miss their deadlines. However, in our case, the ready queue contains the estimate of the execution-time that was used for the schedulability decision. It is then possible, when a task is dispatched, to start a timer that will generate an interrupt at the time when the task would be finished. If the task is not finished at this time, the scheduler may look at (update) the minimum slack for this task. If it is zero, the task is to be terminated but if it is above zero, the scheduler may prolong the execution with at most $s_{\text{pos}(1)}$ time units. It must also be remembered to update the ready queue as if a new task with $A_i = t$, $D_i = d_{\text{pos}(1)} - t$ and $C_i = s_{\text{pos}(1)}$ has arrived. Hence, our admission-controller offers not only overrun protection but also enables less pessimistic estimates of the execution-time.

5. Simulations

The purpose of our simulations is to show how much better our exact admission-controller is compared to the utilization-based admission-controller in [6] in terms of average real-utilization.[7] To make the comparison fair, we have not only considered the result of the admission decision but also the time to make the decision. This is done by assuming that the utilization-based admission-controller takes no time at all while the exact admission-controller uses $\lceil \log |\text{queue}| \rceil + 1$ time units in the worst case. That is, in the admission test this time must be accounted for. If C_A is the worst-case execution-time of the admission-controller, the test becomes $e_{\text{pos}(k+1)} + C_i + C_A \leqslant d_i$ and $s_{\text{pos}(k-1)} - C_A \geqslant C_i$. Furthermore, if $C_A > \min\{s_{\text{pos}(1)}, d_{\text{pos}(1)} - e_{\text{pos}(1)} - C_{\text{pos}(1)}\}$ the admission-controller cannot run

[7] We have chosen not to compare with server-based approaches because they require tuning of server task parameters which is beyond the scope of this paper.

at all since it may cause accepted tasks to miss deadlines. In this case all arriving task are rejected. During admission-control the actual run-time $c_A = |\text{path}|$ will be measured and added to the global variable x which is also introduced in the tests as $e_{\text{pos}(k+1)} + C_i + C_A + x \leqslant d_i$, $s_{\text{pos}(k-1)} - C_A - x \geqslant C_i$ and $C_A > \min\{s_{\text{pos}(1)}, d_{\text{pos}(1)} - e_{\text{pos}(1)} - C_{\text{pos}(1)}\} - x$. When the processor becomes idle x is reset to zero. No overhead is assumed for removing tasks since this is more or less the same for the two approaches.[8]

Example. With this kind of overhead tasks, τ_3, τ_7, τ_8, τ_9 and τ_{10} from Fig. 2 will be rejected, resulting in a utilization of 85%. If the overhead is increased by a factor of five, only τ_1 and τ_2 will be accepted, similar to the utilization-based test.

5.1. Experimental setup

We have used randomly generated tasks sets with varying parameters. The experiments are performed for different *offered load*, $U_{\text{offered}} = \frac{\sum_{i=1}^{n} C_i}{T}$ where T is the length of the experiment. We selected T such that the number of tasks in each task set is in the order of 1000 for $U_{\text{offered}} = 1.0$. Thus, the overhead C_A (and c_A) will be approximately 10 units in the worst case. We then measure the real utilization of a task set as $U_{\text{real}} = \frac{\Gamma}{T}$. (Recall that $\Gamma = \sum C_i$ for all accepted tasks.) The offered load is generated in steps of 0.1 and each value for the real utilization is the average over 100 task sets. The arrival times of the tasks are exponentially distributed while the execution-times and relative deadlines are generated with uniform distribution. If $C_i > D_i$ new values are generated.

5.2. Experimental results

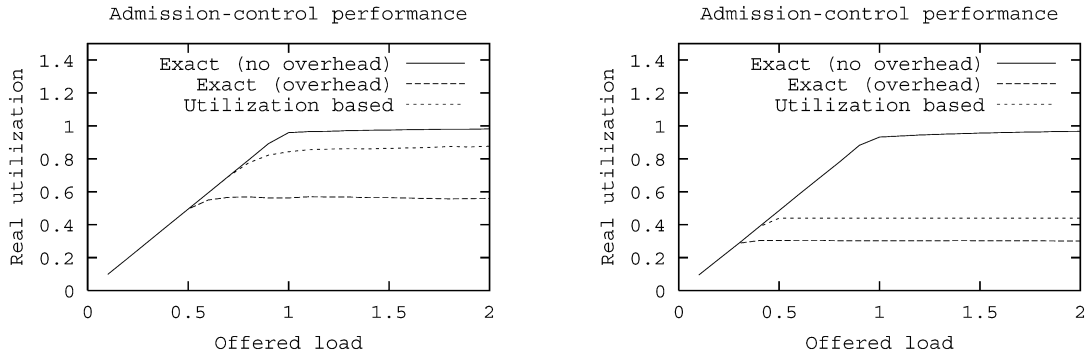
The resulting average utilization for the different admission-controllers for varying values on D_i and C_i is shown for liquid tasks in Fig. 7 and non-liquid tasks in Fig. 8. (We say, based on [11], that a set of aperiodic tasks is liquid if $C_i \ll D_i$.) As can be seen, when overhead is neglected, our exact admission-controller significantly increases the utilization over the sufficient admission-controller. The most drastic improvement occurs when tasks are small and have long deadlines (Fig. 7(b)). The reason is that it is then easy to fit many tasks on the processor but since the utilization-based controller requires that finished tasks remain in the system until their deadlines are expired, these tasks will require a large part of the capacity although they no longer need it. However, when we add overhead, the exact admission-controller actually performs worse than the utilization based (Fig. 7). This is because the inter-arrival times in this case are rather short, causing the admission-controller to run very frequently which results in that accepted tasks get very little processing time. Thus, the ready queue constantly grows (since new tasks are being admitted faster than old ones get executed) which further increases the run-time of the admission-controller. Eventually, when there is not enough slack to run the admission-controller, arriving tasks will be rejected regardless of their parameters. (In this context it is worth remembering that no overhead is assumed for the utilization-based approach which makes it favored towards the exact admission-controller with overhead.) As expected, when tasks are larger (Fig. 8) and inter-arrival times longer, we see that the overhead does not degrade the performance of the exact admission-controller at all. From the plots it can also be seen that, when the tasks have a high utilization, the exact admission-controller performs comparably better and the effect of the overhead is decreased.

In summary, our experiments have shown that, if tasks are not too small, the exact admission-controller outperforms the utilization based even when rather pessimistic overhead is assumed. Hence, the overestimation of the requested capacity by the simpler approach, has a much larger negative impact on the performance than the extra overhead required by the exact method.

6. Extension to periodic tasks

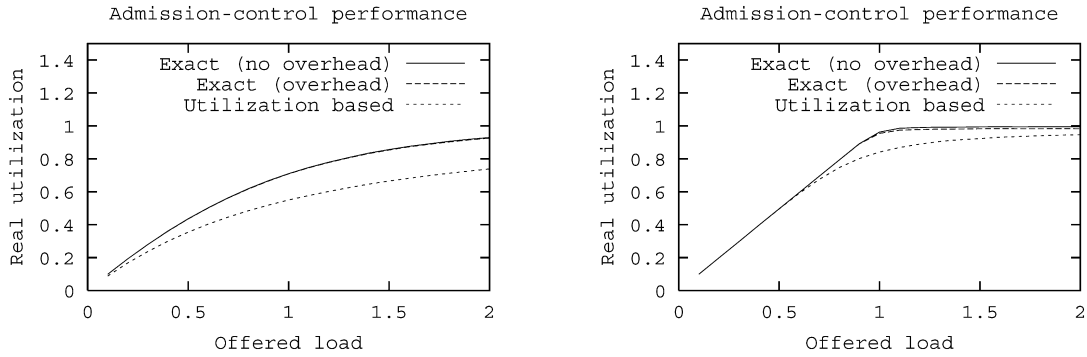
We will now remove the assumption that $\mathcal{T}_{\text{per}} = \emptyset$. When the synthetic utilization is used for admission-control, extension to periodic tasks is trivial since in fact the aperiodic tasks are treated as periodic. Similarly, in our case it is easy to believe that periodic tasks simply can be treated as a number of aperiodic ones. However, this is not the case

[8] The simulator code (in C) is available at http://www.ce.chalmers.se/~cekelin/avl.c.



(a) $D_i \in [500 \ldots 1500)$. (b) $D_i \in [5000 \ldots 15000)$.

Fig. 7. Liquid tasks. $C_i \in [5 \ldots 15)$.



(a) $D_i \in [500 \ldots 1500)$. (b) $D_i \in [5000 \ldots 15000)$.

Fig. 8. Non-liquid tasks. $C_i \in [250 \ldots 750)$.

since we must ensure that *all* future invocations of the periodic tasks will meet their deadlines. That is, the admission-controller for aperiodic tasks must ensure that there is enough capacity left for safe execution of the periodic ones. Therefore, in an exact admission-controller we must distinguish aperiodic tasks from periodic ones.

6.1. Basic idea

The fact that the utilization of the periodic tasks is U_{per} means that over the *hyperperiod* ($\text{lcm}(T_1, \ldots, T_m)$) they require $\text{lcm}(T_1, \ldots, T_m) \cdot U_{\text{per}} = C_{\text{per}}$ time units of computation. This means that there are $\text{lcm}(T_1, \ldots, T_m) - C_{\text{per}} = C_{\text{slack}}$ time units remaining to be used by aperiodic tasks. However, to guarantee the deadlines of the periodic tasks, this slack cannot be arbitrarily distributed. To find the exact slack distribution we can examine the EDF schedule produced by the periodic task and record the slack appearances. Now, since we do EDF scheduling, we know that, if the processor is idle at time t this means that all tasks with $A_i^k < t$ has finished. But this also means that (due to $D_i = T_i$) some of these tasks could have been safely delayed. (We know that $d_i^k > t$ since if $d_i^k \leqslant t$ invocation τ_i^{k+1} would claim the processor.) Thus, slack (or processor idle) indicates that tasks have finished sooner than necessary. In [12], Chetto–Chetto exploit this property in the earliest-deadline-late (EDL) algorithm which schedules tasks as late as possible. Chetto–Chetto then propose an algorithm for calculating a table (or list) containing the location and duration of slack, over the hyperperiod, when EDL is used. We can then recognize that, if we execute a periodic task during a time interval that according to this table contain idle time (slack), the task is executed sooner than necessary and thus the slack will be postponed but not consumed. (It will appear after the task instead of before as the table dictates.) In contrast, if the processor is idle, slack is consumed. Thus, by keeping track of when slack is used compared to when it is available it is possible to know whether aperiodic tasks can be admitted or not with respect to the periodic tasks.

Our admission-controller will be divided into two parts. First we check that the aperiodic tasks, as a whole, will not use more slack than available. Here we will use the slack table from Chetto–Chetto. When the periodic tasks are known to be safe, we check whether the (already admitted) aperiodic tasks also will be safe. Here we will use our previous admission-controller with some extensions.

6.2. Admission-control: periodic tasks

The slack table generated by the algorithm proposed by Chetto–Chetto contains p entries representing the time intervals in the EDL schedule which begin with idle times. An entry contains t_i, the start time of the interval, and Δ_i, the duration of the slack, that is, during $[t_i, t_i + \Delta_i]$ the processor will be idle under EDL. According to [12] this table can be computed in $O(N)$ where N is the number of distinct arrivals within the hyperperiod. However, since the table is computed off-line this complexity does not affect our admission-controller. Of course, there must be enough space to store the table. The table size depends on p which is bounded by $\frac{\text{lcm}(T_1,\ldots,T_m)}{2}$ [12]. To speed up the access of the table information, we introduce an additional field $\Omega_i = \sum_{j=1}^{i-1} \Delta_j$ which holds the total amount of idle times in $[0, t_i]$. Furthermore, we assume that the table is represented as an AVL-tree ordered on t_i which enables entry access in $O(\log p)$. In particular, we will use the operation $\Omega(t)$ which computes the amount of slack in $[0, t]$. Since all table values are restricted to the hyperperiod, we use that,

$$t_{\text{lcm}} = \begin{cases} t \bmod \text{lcm}(T_1,\ldots,T_m), & \text{if } t \bmod \text{lcm}(T_1,\ldots,T_m) \neq 0, \\ \text{lcm}(T_1,\ldots,T_m), & \text{if } t \bmod \text{lcm}(T_1,\ldots,T_m) = 0, \end{cases}$$

and $\Delta_{\text{lcm}} = (t \operatorname{div} \text{lcm}(T_1,\ldots,T_m)) \cdot C_{\text{slack}}$. That is, t_{lcm} is the value to use when looking up in the slack table and Δ_{lcm} is the amount of slack for previous hyperperiods. We then have that $\Omega(t) = \Omega_i + \Delta_{\text{lcm}} + \min(t_{\text{lcm}} - t_i, \Delta_i)$ where $t_i \leqslant_{\max t_i} t_{\text{lcm}}$. Due to the tree implementation, the entry i can be located in $O(\log p)$ and thus $\Omega(t)$ can be computed with the same complexity. Note that the table only contains static information.

In order to record the actual use of the slack we use the counter ω (initially zero) which represents the total amount of consumed (or allocated) slack. Whenever the processor has been idle for some amount of time, ω is increased by the same amount. We also use $d_{\max}$ to represent the largest deadline of the admitted aperiodic tasks. When an aperiodic task τ_i arrives, the admission-control regarding periodic tasks consists of the following test:

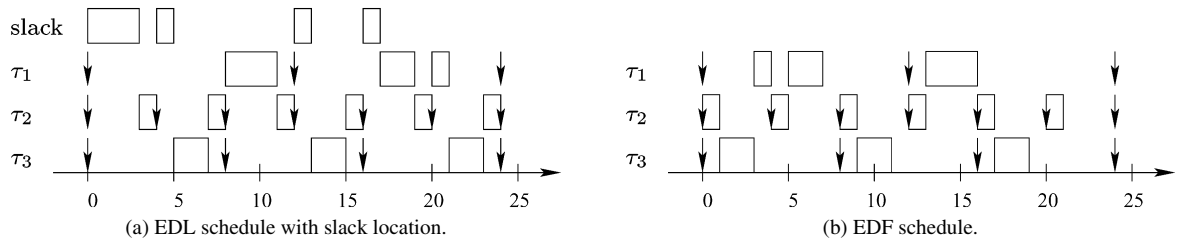
Periodic condition $\Omega(\max(d_{\max}, d_i)) - \omega \geqslant C_i$.

Thus the test considers all aperiodic tasks as one big task and checks whether there is enough processing capacity to allow it to meet its deadline. Our next admission test will determine whether also the individual deadlines of the aperiodic tasks will be met. If the task also passes the aperiodic test, $d_{\max} := \max(d_{\max}, d_i)$ and $\omega := \omega + C_i$.

Example 4. Consider the task set $\mathcal{T}_{\text{per}} = \{\tau_1, \tau_2, \tau_3\}$ where $T_1 = 12$, $C_1 = 3$, $T_2 = 4$, $C_2 = 1$, $T_3 = 8$, $C_3 = 2$. The EDL schedule and the location of the slack for this task set is shown in Fig. 9. The EDF schedule for the same task set is shown in Fig. 9. The slack table contains four entries:

i	0	1	2	3
t_i	0	4	12	16
Δ_i	3	1	1	1
Ω_i	0	3	4	5

Now assume that at $t = 5$ τ_4 arrives with $d_4 = 15$ and $C_4 = 4$. The admission test then becomes $\Omega(15) - 0 \geqslant 4$ and since $\Omega(15) = 4 + 0 + \min(15 - 12, 1) = 5$ the task is accepted and $d_{\max} = 15$, $\omega = 4$. If then at $t = 6$ τ_5 arrives with



(a) EDL schedule with slack location. (b) EDF schedule.

Fig. 9. Example 4 task set.

$d_5 = 12$ and $C_5 = 2$ we see that $\Omega(15) - \omega = 5 - 4 = 1$ which means that this task is rejected. If instead $C_5 = 1$ the task passes this periodic and we must turn to the aperiodic admission test for a complete decision.

6.3. Admission-control: aperiodic tasks

Here we use our previously proposed admission-controller based on the AVL-tree. We will assume that the ready queue contains both admitted aperiodic tasks and released invocations of periodic tasks. To make the admission test correct also in the presence of periodic tasks, we must consider the processing capacity required by future periodic task invocations. Hence, for a task τ_j potentially located at position k in the queue, we use the following term to denote this entity:

- Required processing capacity of future periodic arrivals preceding $\tau_{\text{pos}(k)}$,

$$C(k-1,k) = \sum_{\forall \tau_i \in \mathcal{T}^k_{\text{not\_me}}} \left(\left\lfloor \frac{d_{\text{pos}(k)}}{T_i} \right\rfloor - \left\lfloor \frac{d_{\text{pos}(k-1)}}{T_i} \right\rfloor \right) \cdot C_i,$$

  where $\mathcal{T}^k_{\text{not\_me}}$ is defined in the following way:

  If $\tau_{\text{pos}(k)}$ is a released invocation of a periodic task then
  let $\tau^p_{\text{pos}(k)}$ be the periodic task corresponding to the
  aperiodic task $\tau_{\text{pos}(k)}$
  $\mathcal{T}^k_{\text{not\_me}} = \mathcal{T}_{\text{per}} \setminus \{\tau^p_{\text{pos}(k)}\}$
  otherwise $\tau^p_{\text{pos}(k)}$ is not an instance of a periodic task then
  $\mathcal{T}^k_{\text{not\_me}} = \mathcal{T}_{\text{per}}$.

  Comment: If $k = 1$ then $d_{\text{pos}(k-1)}$ is replaced with A_j.

This rather complicated definition of $C(k-1,k)$ is due to that it, together with the e and s values, is used for both aperiodic and periodic tasks while the admission test is only performed for aperiodic tasks. Informally, $C(k-1,k)$ represents those periodic tasks that will execute between τ_j and its preceding task. Note that it is not necessarily the case that all these tasks actually *do* execute before τ_j since it may finish before they are invoked. In the worst case, the total amount of computation that must take place before τ_j is $e_j = \sum_{l=1}^{k-1}(C_{\text{pos}(l)} + C(l-1,l))$. Thus the minimum slack is computed as $s_j = \min\{d_{\text{pos}(l)} - e_{\text{pos}(l)} - C_{\text{pos}(l)} - C(l-1,l)\colon \forall l > k\}$. This implies that in Condition 1 we cannot use the value $e_{\text{pos}(k+1)}$ since it includes periodic tasks that will execute after the task considered for admission-control. Instead we have:

- *Aperiodic condition* 1. $e_{\text{pos}(k-1)} + C_{\text{pos}(k-1)} + C_j + C(k-1,k) \leqslant d_j$.
- *Aperiodic condition* 2. $s_{\text{pos}(k-1)} \geqslant C_j$.

Note that Condition 2 is unaffected since the periodic tasks that was added for Condition 1 are accounted for already. The inserted task gets the values $e_j = e_{\text{pos}(k-1)} + C_{\text{pos}(k-1)} + C(k-1,k)$ and $s_j = s_{\text{pos}(k-1)} - C_j$. If τ_j is an aperiodic task the procedure for updating the values for succeeding tasks does not change, i.e., the update note is still $s_{\text{set}} = -1, e_{\text{inc}} = C_j$. For preceding tasks the note becomes $s_{\text{set}} = \min\{d_j - e_j - C_j - C(k-1,k), s_j\}, e_{\text{inc}} = 0$. However, if τ_j is a released invocation of a periodic task then its computation demand has been accounted for already and hence no updates of succeeding tasks are required. For preceding tasks, an update of the minimum slack is required if the new task also is the one with the minimum slack.

As mentioned, the scheduled start time e may now be greater than the actual start time. However, this poses no problem since when the processor becomes idle the scheduler simply picks the first task in the ready queue. The total amount of computation, as accounted for by succeeding tasks, will be the same regardless of the task order.

The computation of $C(k-1,k)$ requires that all periodic tasks are examined and thus the complexity of this second admission test (and insertion in the ready queue) becomes $O(m + \log n)$.

Example 5. We consider example 4 again. We assume that τ_1^1 is the first to arrive. Then its accumulated execution-time of preceding tasks would be: $e_1^1 = C(0,1) = (\lfloor \frac{12}{4} \rfloor - \lfloor \frac{0}{4} \rfloor) \cdot 1 + (\lfloor \frac{12}{8} \rfloor - \lfloor \frac{0}{8} \rfloor) \cdot 2 = 3 + 2 = 5$ since there are no preceding tasks in the queue. We assume τ_2^1 to be the next task to arrive. Since $d_2^1 < d_1^1$ it will be the first in the queue with $e_2^1 = C(0,1) = (\lfloor \frac{4}{12} \rfloor - \lfloor \frac{0}{12} \rfloor) \cdot 3 + (\lfloor \frac{4}{8} \rfloor - \lfloor \frac{0}{8} \rfloor) \cdot 2 = 0 + 0 = 0$ and $s_1^2 = 4$. Since $d_2^1 < d_3^1 < d_1^1$, τ_3^1 will be inserted between τ_2^1 and τ_1^1 with $C(1,2) = (\lfloor \frac{8}{12} \rfloor - \lfloor \frac{4}{12} \rfloor) \cdot 3 + (\lfloor \frac{8}{4} \rfloor - \lfloor \frac{4}{4} \rfloor) \cdot 1 = 0 + 1 = 1$ which gives $e_3^1 = 0 + 1 + 1 = 2$ and $s_3^1 = 4$. Note that all e and s values will be the same regardless of the order in which the tasks are handled.

When τ_2^2 arrives at $t = 4$ only τ_1^1 remains in the queue. The values for τ_1^1 are updated as $e_1^1 := 5 + 1 = 6$ and $C_1^1 := 3 - 1 = 2$ and τ_2^2 is inserted at position 1 with $e_2^2 = 4$ and $s_2^2 = 4$. At $t = 5$, τ_1^1 again will be the only task in the queue and τ_4 will be located at position 2 in the queue since $d_4 > d_1^1$. Thus $C(1,2) = (\lfloor \frac{15}{12} \rfloor - \lfloor \frac{12}{12} \rfloor) \cdot 3 + (\lfloor \frac{15}{4} \rfloor - \lfloor \frac{12}{4} \rfloor) \cdot 1 + (\lfloor \frac{15}{8} \rfloor - \lfloor \frac{12}{8} \rfloor) \cdot 2 = 0$ which gives $e_4 = 6 + 2 + 0 = 8$ and $s_1^1 = 15 - 8 - 4 - 0 = 3$. Hence, for τ_5 at $t = 6$ we have that $C_1^1 := 2 - 1 = 1$ and $e_1^1 := 6 + 1 = 7$ and since $d_1^1 = d_5 < d_4$, τ_5 will be located at position 2 in the queue. Thus $C(1,2) = 0$ again and both Conditions 1 and 2 succeed since $7 + 1 + 1 + 0 \leqslant 12$ and $3 \geqslant 1$. Which means that τ_5 is added with $e_5 = 7 + 1 + 0 = 8$ and $s_5 = 3 - 1 = 2$. The update notes result in $e_4 = 9$ and $s_1^1 = 2$. Note that the minimum slack now appears to be 2 although in reality it is zero due to the periodic tasks that not yet has arrived but later on will be postponed by the aperiodic tasks.

6.4. Overall admission-control

By combining the computational complexity of the admission test for the periodic tasks ($O(\log p)$) with the complexity of the test for already admitted aperiodic tasks ($O(m + \log n)$) the overall computational complexity of the admission-controller becomes $O(m + \log n + \log p)$. Although this complexity may appear discouragingly high, it should be remembered that on rejection the complexity may be much lower. Furthermore, since previous exact admission-controllers require $O(p)$ steps and typically $p \gg m$ our method provides a clear improvement.

7. Related and future work

We have already mentioned the work on utilization-based admission-control [6] which provides sufficient tests for task sets with purely aperiodic tasks as well as for a mixture of periodic and aperiodic tasks. Our algorithms assumed that $R_1 = R_2 = \cdots = R_m$ but utilization-based admission did not, so clearly if the task set does not satisfy $R_1 = R_2 = \cdots = R_m$ then utilization-based admission-control is superior. Exact tests for joint scheduling of periodic and aperiodic have been proposed in [12,13]. However, the runtime complexity of these tests is $O(N)$ where N is the number of slots or arrivals within the hyperperiod. (In addition, [12] assumes that $\mathcal{T}_{\text{aper}} = \emptyset$ whenever an aperiodic task arrives.) The admission-controller in [14] solves a problem very similar to the problem we address. They study exact admission-control of aperiodic tasks (called sporadic tasks in [14]) in the presence of a periodic baseload. However, the computational complexity of their solution is worse. Another popular approach for this joint scheduling is aperiodic servers, e.g., [10]. However, these servers typically assume that aperiodic tasks have no deadlines but rather that their response times are to be minimized. Furthermore, the server parameters must be set according to some anticipated workload which can be hard to predict [15]. Nevertheless, a popular property of aperiodic servers is their ability to provide task isolation through policing. That is, a subset of tasks (constituting an application) is prevented from requesting more than its predetermined share. In this line of work, [16] proposes the BSS algorithm which uses an AVL-tree with lazy updates resembling our approach. The proposed algorithm maintains budgets—similar to our minimum slacks—to decide whether an application may execute or not. (When a budget becomes zero any remaining tasks will not be allowed to execute.) However, the budget calculation does not consider the task execution-times until *after* some task's execution. This means that, in an overload situation, there will still be tasks awaiting execution when the budget is zero. In particular, it is not possible to know when a task *arrives* whether it will be executed or not.

As an example, consider the tasks τ_1 with $d_1 = 10$, $C_1 = 5$ and τ_2 with $d_2 = 9$, $C_2 = 6$. It is assumed that $A_1 = A_2 = 0$. With our admission-controller τ_2 will be rejected at arrival which is the expected result. However, using the BSS algorithm both tasks will be admitted with budgets B_i equaling their deadlines. τ_2 will then be the first to execute and when it finishes, the budget of τ_1 will be decreased by C_2 such that $B_1 = 10 - 6 = 4$. τ_1 will then be

allowed to execute until its budget is exhausted. Thus, since $C_1 > B_1$ the task will not finish within its deadline. From this example we can note two things: (i) an admitted task is not guaranteed to meet its deadline and (ii) the budget information is not enough to make a correct admission decision. Hence, the proposed data structure and update scheme of BSS cannot trivially be applied to admission-control.

The on-line scheduling algorithm D* [9] also resembles our admission-controller in the data structure it uses. However, in D* an (accepted) task is never guaranteed to finish since it may be rejected later on due to "better" tasks coming in, i.e., tasks which contribute more to the cumulative value. As with BSS, D* is not trivially extended to an admission-controller since the information that it maintains is not sufficient to perform an admission test.

Our admission-controller is straightforward to extend to multiprocessor systems. Given some order of the processors, when a task arrives, it is simply shipped to the first processor, if this processor cannot accept the task, it is instead shipped to processor two and so on until the task is either accepted or all processors have been considered. Hence, the admission-controller would run in $O(m \cdot \log n)$. If response time of the task is not an issue, it is generally best to select processors in a first-fit order [17] since this prevents the so-called Dhall's effect [18].

An issue that has not been considered in previous research is in what order tasks with the same arrival time should be passed to the admission test. In this paper we have assumed that the order is given, i.e., if $A_i = A_j$ the cumulative value may differ substantially depending on how the choice is made. If the deadlines differ it may seem reasonable to consider tasks with shorter deadlines first although this is not necessarily optimal. For example, if many such tasks with short deadlines are rejected, the time spent by the admission-controller on these tasks may cause a task with longer deadline to be rejected as well. Hence, in this situation it can be fruitful to use the QoS information to allow rejection/acceptance of several tasks simultaneously. In particular, if the deadlines are equal, this information enables us to make the optimal choice by solving the corresponding so-called subset-sum problem. Unfortunately this problem is NP-hard but a pseudo-polynomial algorithm exists that runs in $O(n_{\text{sim}} \cdot c^2)$ where n_{sim} is the number of tasks that arrive simultaneously and c is the available capacity [19].

8. Conclusions

In this paper we propose an exact admission-control algorithm for aperiodic tasks and EDF. For purely aperiodic task sets our algorithm runs in $O(\log n)$. This is the same run-time complexity as for utilization-based admission-controllers that only provide a sufficient admission test. Although our algorithm has a larger (constant) overhead, our experiments show that, if tasks are not too small, the average utilization is yet higher than for simpler approaches. In addition, the information maintained by our admission-controller can be used to achieve a number of other features such as task isolation and capacity reclamation. For applications with a mix of aperiodic and periodic tasks, our admission-controller runs in $O(m + \log n + \log p)$ which is considerably faster than previous approaches that run in $O(p)$ since typically $p \gg m$.

References

[1] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. Assoc. Comput. Mach. 20 (1) (January 1973) 46–61.
[2] B. Andersson, Static-priority scheduling on multiprocessors, PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2003.
[3] T. Abdelzaher, C. Lu, Schedulability analysis and utilization bounds for highly scalable real-time services, in: Proc. of the IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan, May 30–June 21, 2001, pp. 15–25.
[4] B. Andersson, T. Abdelzaher, J. Jonsson, Partitioned aperiodic scheduling on multiprocessors, in: Proc. of the IEEE International Parallel and Distributed Processing Symposium, Nice, France, April 22–26, 2001.
[5] D.E. Knuth, The Art of Computer Programming, vol. 3. Sorting and Searching, Addison–Wesley, Reading, MA, 1975.
[6] T.A. Abdelzaher, C. Lu, A utilization bound for aperiodic tasks and priority driven scheduling, IEEE Trans. Comput. 53 (3) (March 2004) 334–350.
[7] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo, Fundamentals of EDF scheduling, in: Deadline Scheduling for Real-Time Systems, Kluwer Academic, Boston, 1998, pp. 27–65 (Chapter 3).
[8] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, Englewood Cliffs, NJ, 1976.
[9] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, On-line scheduling in the presence of overload, in: Proc. of the IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, October 1991, pp. 100–110.
[10] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proc. of the IEEE Real-Time Systems Symposium, Madrid, Spain, December 2–4, 1998, pp. 4–13.

[11] T.F. Abdelzaher, B. Andersson, J. Jonsson, V. Sharma, M. Nguyen, The aperiodic multiprocessor utilization bound for liquid tasks, in: Proc. of the IEEE Real-Time Technology and Applications Symposium, San José, California, September 25–27 2002, pp. 173–186.
[12] H. Chetto, M. Chetto, Some results of the earliest deadline scheduling algorithm, IEEE Trans. Softw. Engrg. 15 (10) (October 1989) 1261–1269.
[13] G. Fohler, Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems, in: Proc. of the IEEE Real-Time Systems Symposium, Pisa, Italy, December 5–7, 1995, pp. 152–161.
[14] T.-S. Tia, J.W.-S. Liu, J. Sun, R. Ha, A linear-time optimal acceptance test for scheduling of hard real-time tasks, Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1994; available at the URL: http://www-rtsl.cs.uiuc.edu/papers/TiLS94c.ps.
[15] L. Abeni, L. Palopoli, G. Lipari, J. Walpole, Analysis of a reservation-based feedback scheduler, in: Proc. of the IEEE Real-Time Systems Symposium, Austin, Texas, December 3–5 2002, pp. 71–80.
[16] G. Lipari, S. Baruah, Efficient scheduling of real-time multi-task applications in dynamic systems, Proc. of the IEEE Real-Time Technology and Applications Symposium, Washington, DC, USA, May 31–June 2, 2000, pp. 166–175.
[17] D.S. Johnson, Near-optimal bin-packing algorithms, PhD thesis, Massachusetts Institute of Technology, 1974.
[18] B. Andersson, S. Baruah, J. Jonsson, Static-priority scheduling on multiprocessors, in: Proc. of the IEEE Real-Time Systems Symposium, London, UK, December 3–6, 2001, pp. 193–202.
[19] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, Wiley, Chichester, 1990.