A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling *

Tarek F. Abdelzaher, Vivek Sharma
Department of Computer Science, University of Virginia, Charlottesville, VA 22904

Chenyang Lu
Department of Computer Science, Washington University, St. Louis, St. Louis, MO 63130

Abstract

Real-time scheduling theory offers constant-time schedulability tests for periodic and sporadic tasks based on utilization bounds. Unfortunately, the periodicity or the minimal interarrival-time assumptions underlying these bounds make them inapplicable to a vast range of aperiodic workloads such as those seen by network routers, web servers, and event-driven systems.

This paper makes several important contributions towards real-time scheduling theory and schedulability analysis. We derive the first known bound for schedulability of aperiodic tasks. The bound is based on a utilization-like metric we call synthetic utilization, which allows implementing constant-time schedulability tests at admission control time. We prove that the synthetic utilization bound for deadline-monotonic scheduling of aperiodic tasks is \( \frac{1}{1 + \sqrt{1/2}} \). We also show that no other time-independent scheduling policy can have a higher schedulability bound. Similarly, we show that EDF has a bound of 1 and that no dynamic-priority policy has a higher bound. We assess the performance of the derived bound and conclude that it is very efficient in hit-ratio maximization.

Keywords: Real-time scheduling, schedulability analysis, utilization bounds, aperiodic tasks.

1 Introduction

A fundamental problem in real-time scheduling is that of computing the schedulability of a task set. For periodic and sporadic tasks, sufficient schedulability conditions exist that relate schedulability to

---

*The work reported in this paper was supported in part by the National Science Foundation under grants CCR-0093144, CCR-0208769, and ANIR-0105873, and under MURI grant N00014-01-1-0576.
aggregate utilization. The best-known result in this area is the original utilization bound derived by Liu and Layland [18] for periodic tasks. To date, no such bound exists for aperiodic tasks. In this paper, we prove for the first time that aperiodic tasks with arbitrary arrival times, computation times and deadlines are schedulable by a time-independent policy if their aggregate synthetic utilization\(^1\) does not exceed \(\frac{1}{1 + \sqrt{1/2}}\). (We also trivially show that for dynamic-priority scheduling, the bound is 1.)

Our work is motivated by the timing requirements of high-performance real-time services such as online trading. High performance servers may need to handle tens of thousands of requests per second. If the service is real-time, it must ensure that all requests are served within their respective deadlines. While mainstream web services do have real-time applications, the primary market of the developers of such systems remains predominantly best-effort. Vendors whose primary market is best effort applications are generally reluctant to embed polynomial time real-time schedulability analysis tests in their products. A constant-time test that promises significant added value in terms of timeliness guarantees is much more appealing. Such a test is more likely to be widely deployed in mainstream applications in practice, which is the primary motivation for the approach taken in this paper. Hence, we derive an efficient online test that determines the schedulability of incoming tasks based on simple inspection of the current value of a utilization-like metric. Consequently, a constant-time admission control can be applied to ensure that the deadlines of all admitted tasks are met, and maximize the admitted task ratio, while making no assumptions regarding the task arrival pattern.

For portability considerations we are more interested in deriving schedulability conditions under static scheduling policies in which task invocation priorities do not depend on absolute time. Dynamic priority policies such as EDF, while addressed in this paper, are not implemented on certain standard server operating systems, thereby limiting the applicability of their results.

Deriving a schedulable bound for aperiodic tasks is a conceptual shift that calls for extending some basic notions in rate-monotonic analysis to apply to the addressed problem. More specifically, we use the notion of synthetic utilization\(^2\) – a time varying quantity that depends on current arrivals and deadlines. It reduces to Liu and Layland’s definition of utilization factor when analyzing (a critical instant of) periodic tasks. We show that the actual measured processor utilization of admitted tasks is lower-bounded by its synthetic utilization at all times. Synthetic utilization is used for admission control.

---

1. Synthetic utilization is defined in Section 3.
2. Also known as instantaneous utilization.
We also define a generalized notion of fixed-priority scheduling that applies to aperiodic tasks, where each task has only one invocation. Specifically, we consider a category of scheduling policies we call time-independent, in which individual invocation priorities are independent of invocation arrival times. These policies are especially suitable for abstracting differentiated services scheduling algorithms. Using these notions, we make two main contributions to real-time scheduling theory:

- We prove that deadline-monotonic scheduling is the optimal time-independent scheduling policy for aperiodic tasks in the sense of maximizing the utilization bound. Observe that optimality is defined here in the same sense used when calling rate-monotonic scheduling an optimal fixed-priority scheduling policy for periodic tasks. It implies that no other scheduling policy in its class can have a higher schedulability bound. An admission controller based on this bound, however, may reject some tasks unnecessarily.

- We prove that the bound for schedulability of aperiodic tasks under deadline-monotonic scheduling is \( \frac{1}{1+\sqrt{1/2}} \) (i.e., 58.6\%). The bound is lower than the Liu and Layland bound for periodic tasks. This is because by relaxing the task periodicity requirement, we consider a larger set of possible task arrival patterns. The minimum utilization of an unschedulable pattern in that larger set is smaller than the minimum utilization in the subset of task arrival patterns that are periodic (the Liu and Layland bound).

Schedulable utilization bounds are, by construction, pessimistic. Exceeding the bound does not necessarily entail deadline misses. We merely provide a sufficient schedulability condition that has the merit of being computationally efficient. What we show, however, is that our bound is tight in that there exists an unschedulable pattern of aperiodic task arrivals whose synthetic utilization is infinitesimally above \( \frac{1}{1+\sqrt{1/2}} \).

What is more important, as shown in the evaluation section, is that for a large range of workloads, the actual measured processor utilization can be near saturation even when synthetic utilization does not exceed the bound. This is due to the way synthetic-utilization is defined. Hence, admission control tests based on synthetic utilization do not underutilize the processor.

The remainder of this paper is organized as follows. Section 2 introduces a new task model we call the *acyclic* model. We show that any set of aperiodic tasks can be represented by an equivalent set
of acyclic tasks. The utilization bound for time-independent scheduling of acyclic tasks is described in Section 3. For completeness, Section 4 derives the corresponding bound for fully dynamic-priority scheduling. Section 5 evaluates the practicality of the derived bound. Section 6 describes related work. Finally, Section 7 presents the conclusions of the paper and avenues for future work.

2 The Acyclic Task Model

To simplify the derivation of the utilization bound for aperiodic tasks, in this paper, we first define a more structured task model we call the acyclic model. We then show that the utilization bound derived for this model is identical to that for aperiodic tasks. Hence, while our derivation can be based on the former model, the results apply to the latter.

An acyclic task is composed of a chain of successive task invocations. Each task invocation $T_i$ has an arrival time $A_i$, an execution time $C_i \geq 0$, and a relative deadline $D_i > 0$ (hence, the absolute deadline $d_i$ of the invocation is $d_i = A_i + D_i$). Unlike the case with periodic and sporadic tasks, different invocations of the same acyclic task have independent execution times and deadlines. The deadline of each task invocation is the arrival time of the next invocation of the same task. Figure 1 shows an example of three acyclic tasks. The horizontal timelines represent the progression of individual tasks. The vertical bars delimit successive invocations of the same task. Each bar marks both the deadline of the previous invocation and the arrival time of the next. The shaded rectangles represent invocation execution times. In this figure, the tasks are scheduled as if each had its own processor. Note that some task invocations have a zero execution time, which is allowed in the acyclic task model.

It is easy to see that any set of aperiodic task invocations can be transferred to an equivalent set of acyclic task invocations as follows:

- Partition all aperiodic task invocations into sequences (of one or more invocations each) such that for any two consecutive invocations $T_i$ and $T_j$ in the same sequence, the absolute deadline $d_i$ of the first invocation is no later than the arrival time $A_j$ of the next (i.e., $d_i \leq A_j$). Let the number of such sequences be $n$.

- For every two consecutive invocations $T_i$ and $T_j$ in the same sequence (where $T_i$ precedes $T_j$), if $d_i < A_j$ insert a third “dummy” invocation $T_k$ between $T_i$ and $T_j$ with arrival time $A_k = d_i$, deadline $d_k = A_j$, and execution time $C_k = 0$. Note that the added invocation does not alter
schedulability and does not change utilization. When this process is completed, each of the \( n \)
invocation sequences thus formed satisfies the definition of an acyclic task.

The above transformation suggests that since for each aperiodic arrival pattern there exists an equivalent
acyclic arrival pattern of the same utilization and schedulability status (and, trivially, vice versa), a bound
derived for acyclic tasks will apply to aperiodic tasks as well. Hence, it suffices to derive a utilization
bound for acyclic tasks. This bound will depend on the number of acyclic tasks \( n \) (which is why it is
easier to express it for an acyclic task model, rather than directly for an aperiodic model). Since in high
performance servers a large number of requests are served concurrently, the number of acyclic tasks \( n \) is
typically very large. Therefore, for all practical purposes, we are most interested in the asymptotic value
of the bound as \( n \to \infty \). This asymptotic value can be used by an admission controller facing aperiodic
tasks without the need to translate the incoming invocation pattern into an equivalent set of acyclic tasks.

3 The Generalized Bound for Acyclic Tasks

Consider the simple model of scheduling independent acyclic tasks on a uniprocessor. We begin by
introducing the concept of synthetic utilization, which is central to the derivation of our schedulability
bound. At any given time instant, \( t \), let \( V(t) \) be the set of all task invocations that have arrived but whose
deadlines have not expired, i.e., \( V(t) = \{T|A_i \leq t < A_i + D_i\} \). Let \( n(t) \) be the number of these
invocations at time \( t \). In the acyclic task model, \( n(t) \) remains constant over time, \( n(t) = n \) (although
some invocations may have zero execution times). Figure 2 demonstrates the set \( V(t) \) (shown in black)
at a randomly chosen time instant, \( t_0 \), for the three acyclic tasks of Figure 1. The intervals of busy and
idle CPU time are shown at the bottom of the figure.

![Figure 1. An Example Acyclic Task Set](image1.png)

![Figure 2. \( V(t_0) \): Task invocations at \( t_0 \)](image2.png)
Let a schedule gap be a period of idle CPU time. Observe that all task invocations that arrive before any such gap have no effect on the schedule after the gap. Hence, it is useful to define the subset $S(t) \subseteq V(t)$ which includes only those task invocations in $V(t)$ that arrive after the beginning of the last schedule gap. We call them current invocations. For example, at time $t_0$ in Figure 2, the set of current invocations, $S(t_0)$, would include only those of task 2 and task 3. It would exclude the shaded invocation of task 1 since that invocation arrives before a schedule gap that precedes $t_0$. Observe that the definition of the set of current invocations $S(t)$ does not depend on the actual completion time of different invocations. Invocation completion times are scheduling-policy-dependent. In contrast, we define $S(t)$ in a manner that is independent of the particular scheduling policy used. Note also that, by definition, $S(t)$ is empty during a schedule gap. Synthetic utilization, $U(t)$, is defined as the utilization contributed by the current task invocations at time $t$, given by the expression:

$$U(t) = \sum_{T_i \in S(t)} C_i / D_i$$

(1)

A simple counter can keep track of synthetic utilization at run-time. The counter would be reset to zero at the beginning of each schedule gap. It would be incremented by $C_i / D_i$ upon the admission of a new task invocation $T_i$, and decremented by $C_i / D_i$ when the deadline of a current invocation $T_i$ is reached. The set of current invocations is emptied when a schedule gap is encountered. A pseudo-code for the utilization-based admission controller is shown in Figure 3. The initialization() procedure is executed at the beginning. The request_handler$(C_i, D_i)$ and gap_handler() are executed upon the arrival of a new task invocation and the beginning of a new schedule gap respectively. Finally, event $(i)$ is called by the operating system timer.$^3$

We shall prove that using an optimal time-independent scheduling policy, all task invocations will meet their deadlines if $\forall t : U(t) \leq UB(n)$, where $UB(n) = \frac{1}{2} + \frac{1}{2n}$ for $n < 3$ and $UB(n) = \frac{1}{1 + \sqrt{\frac{1}{2}(1 - \frac{1}{n})}}$ for $n \geq 3$. (At $n = 3$ the two expressions are equivalent.) When the number of current invocations, $n$, increases, the bound approaches $\frac{1}{1 + \sqrt{1/2}}$.

It is important to notice that the aforementioned expression of synthetic utilization reduces to the expression of utilization used in prior schedulability bound literature when tasks are periodic. The periodic

---

$^3$In practice, it is more efficient to execute pending events only when a new invocation arrives, hence avoiding the need for fine-grained timers.
task analysis is invariably conducted for a critical instant of the task set [18] (which contains no schedule gaps). During that interval, exactly one invocation is current from every task in the periodic task set. Hence, $S(t)$ is constant and includes a single invocation of each task in the set, i.e., synthetic utilization is reduced to the standard expression of the utilization factor as defined by Liu and Layland [18]. Due to this equivalence, the value of the utilization bound derived in this paper can be meaningfully compared to the bound by Liu and Layland and its extensions for the different periodic task models.

### 3.1 Synthetic versus Measured Utilization

In the case of aperiodic tasks, synthetic utilization, $U(t)$, is not necessarily equal to the measured utilization of a processor, defined as the fraction of time it is not idle. The two are different because the expression for $U(t)$ takes into consideration invocation deadlines, whereas measured utilization does not. Consider the case where admission control or some other mechanism such as QoS adaptation prevents synthetic utilization $U(t)$ from exceeding a ceiling that is less than 1. This is the case, for example, with the admission controller in Figure 3.

Under the above condition, it is easy to show that the synthetic utilization, $U(t)$, of admitted tasks, averaged over any time interval is less than the measured utilization of the processor averaged over the same time interval. To see why, observe that the actual instantaneous utilization of the processor is
either zero, within a schedule gap, or 1, when some task is running. In contrast, the synthetic utilization, $U(t)$, is zero within a schedule gap (since $S(t)$ is empty), but is upper-bounded by a value less than 1 when a task is running. Hence, it is always less than the measured utilization (unless both are zero). This is true of both instantaneous values and averages over an arbitrary time window. The property is very important because it means that satisfying the schedulability bound on the synthetic utilization $U(t)$ does not necessarily imply underutilizing the processor. In fact, in the evaluation section we show that the processor can operate near full capacity while the schedulability bound on $U(t)$ is observed by admission control. In the rest of this paper, when we refer to utilization we mean synthetic utilization, unless explicitly stated otherwise.

3.2 Time-Independent Scheduling

Traditionally, scheduling policies are classified into fixed-priority and dynamic-priority depending on whether or not different invocations of the same task have the same priority. Associating fixed priority levels with acyclic tasks is meaningless because different task invocations can have arbitrarily different deadlines and execution times. Instead, we consider a broader category of scheduling policies which we call time-independent. A scheduling policy is time-independent if the priority assigned to a task invocation does not depend on its absolute arrival time. Such a policy (i) classifies individual task invocations into a finite number of classes, (ii) associates a fixed priority with each class (i.e., independently of invocation arrival times), and (iii) preemptively schedules tasks of different priorities in priority order while scheduling tasks of the same priority in a FIFO order. Hence, we formally define time-independent scheduling as follows:

**Definition:** A scheduling policy is time-independent if there exists a function $f(\tau, t) \rightarrow P$, such that:

1. $f(\tau, t) \rightarrow P$ maps an infinite set of task invocations $\tau$ whose arrival times are given by vector $t$ into a finite set of priority values $P$.

2. $f(\tau, t) \rightarrow P$ satisfies $f(\tau, t) = f(\tau, t')$, for any $t$ and $t'$.

3. The processor’s ready queue is sorted by values $P$ such that the ready task with the smallest value is resumed next. Tasks with the same value of $P$ are queued in FIFO order.

Note that EDF, for example, is not time-independent by the above definition, since property 1 may be violated. To see why, consider an infinite set of independent task invocations $\tau$, each of which (i) arrives
before the previous one finishes, and (ii) has a shorter absolute deadline. In this case, the priority of the 
\(i^{th}\) arrival must be higher that that of the \((i - 1)^{th}\) arrival, to ensure proper preemption in accordance with 
EDF. Since this relation applies to every pair of successive arrivals, the number of priorities needed must 
be equal to the number of arrivals. An infinite set of such invocations therefore cannot be mapped into 
a finite set of priorities, which violates property 1. Moreover, should these invocations have arrived in a 
different order, their relative priorities could have been different under EDF, depending on the resulting 
order of their absolute deadlines. Hence, property 2 may be violated as well.

In contrast, FIFO is time-independent. In the context of the above definition, FIFO can be thought 
of as a priority-driven policy in which all tasks are assigned the same priority, regardless of the time at 
which they arrive. Hence, there exists a priority assignment which satisfies properties 1 and 2. In this 
assignment, same priority tasks are scheduled FIFO, satisfying property 3. By definition, the existence 
of one priority assignment which satisfies the three properties is sufficient to render the policy time-
independent. A minor point of clarification is that the definition states that only the priority assignment 
should be independent of invocation arrival times. The total order of scheduled invocations is not. 
In particular, the order of same priority invocations depends on their arrival times since same priority 
invocations which arrive earlier are scheduled first.

When \(P\) is small, a time-independent scheduling policy can be easily implemented on current operat-
ing systems that support a fixed (finite) number of priority levels. This, for example, is akin to diff-serv 
arichitectures which classify all network traffic into a finite number of classes and give some classes 
priority over others.

One time-independent classification of tasks would be by their relative deadlines, \(D_i\). For example, an 
online trading server can serve its “gold” customers within a delay bound of 1 second, serve its “silver” 
customers within a delay bound of 5 seconds, while offering non-customers a best-effort service. We call 
a scheduling policy that assigns higher priority to invocations with shorter relative deadlines, a deadline 
monotonic scheduling policy. We shall prove that this policy achieves the highest schedulable bound.

3.3 The Sense of Optimality

In this paper, we call a scheduling policy optimal in a class if it maximizes the schedulable utilization 
bound among all policies in that class. For example, in the case of periodic tasks, no fixed-priority 
scheduling policy has a higher bound than rate-monotonic scheduling. Hence, rate-monotonic schedul-
ing is the optimal fixed-priority policy. In the same sense, EDF is the optimal dynamic priority policy. In the following, we show that deadline-monotonic scheduling maximizes the bound among all time-independent policies of aperiodic tasks. Observe that we are interested in the schedulability of a particular arrival pattern (i.e., a rigid task set).

3.4 The Optimal Utilization Bound

In this section, we first derive the utilization bound for schedulability of acyclic tasks under the deadline monotonic scheduling policy. We then show that no time-independent policy can achieve a higher bound. Let a \textit{critically-schedulable} task pattern be one in which some task invocation has zero slack. The derivation of the utilization bound for deadline-monotonic scheduling undergoes the following steps:

1. Consider the set \( S \) of all possible critically-schedulable task invocation patterns.

2. Find, for each pattern \( \zeta \in S \), the maximum utilization \( U_{\max}^{\zeta} = \max_t U(t) \) that occurs prior to the deadline of a critically-schedulable invocation. By definition of \( U_{\max}^{\zeta} \), there exists at least one point (prior to the deadline of a critically-schedulable invocation) in the pattern where the utilization \( U(t) = U_{\max}^{\zeta} \).

3. Compute the minimum such utilization \( U_{\min} = \min_{\zeta \in S} \{U_{\max}^{\zeta}\} \) across all patterns. Thus, in every critically-schedulable pattern there exists at least one point (prior to the deadline of a critically-schedulable invocation) where the utilization \( U(t) = U_{\max}^{\zeta} \geq U_{\min} \).

\( U_{\min} \) is therefore the utilization bound. Since each unschedulable independent task set (on a uniprocessor) can be made critically-schedulable by reducing the execution time of the invocations that miss their deadlines, there always exists a critically-schedulable set of lower utilization. Hence, any unschedulable invocation pattern will necessarily exceed our derived bound at some point prior to each deadline miss. In a system where the bound is never exceeded it must be that all invocations meet their deadlines.

The key to a succinct derivation of the utilization bound is to avoid explicit enumeration of all possible task patterns in set \( S \) defined in step (1). Let the \textit{worst-case critically-schedulable task invocation pattern} (or worst-case pattern, for short) be defined as one whose utilization never exceeds \( U_{\min} \). We begin by finding a subset of the set of all possible patterns \( S \) proven to contain a worst-case pattern (one with the minimum utilization \( U_{\min} \)). In particular, we prove that the worst-case critically-schedulable task arrival pattern has the following two properties: (i) the lowest priority task invocation is critically-schedulable,
and (ii) the synthetic utilization, $U(t)$, remains constant, say $U^\zeta$, in the busy interval prior to the deadline of the lowest priority invocation (where by busy interval we mean one of continuous CPU consumption). Hence, $U^\zeta_{\text{max}} = U^\zeta$ where $U^\zeta$ is the constant utilization in the aforementioned interval.\(^4\)

Subject to the above properties, we then find an analytic expression for utilization $U^\zeta$ as a function of the parameters of a general invocation pattern $\zeta$. Finally, we minimize $U^\zeta$ with respect to pattern parameters to obtain the utilization bound. Below we present the proof in more detail.

**Theorem 1:** A set of $n$ acyclic tasks is schedulable using the deadline-monotonic scheduling policy if
\[ \forall t : U(t) \leq UB(n), UB(n) = \frac{1}{2} + \frac{1}{2n} \text{ for } n < 3 \text{ and } UB(n) = \frac{1}{1 + \sqrt{n(1 - \frac{1}{n})}} \text{ for } n \geq 3. \]

**Proof:** Let us consider a critically-schedulable acyclic task invocation pattern. By definition, some task invocation in this pattern must have zero slack. Let us call this task invocation $T_m$. Consider the interval of time $A_m \leq t < A_m + D_m$ during which $T_m$ is current. At any time $t$ within that interval, $U(t) = C_m/D_m + \sum_{i:T_i > T_m} C_i/D_i + \sum_{i:T_i < T_m} C_i/D_i$, where $C_m/D_m$ is the utilization of task invocation $T_m$, $\sum_{i:T_i > T_m} C_i/D_i$ is the utilization of higher priority task invocations that are current at time $t$, and $\sum_{i:T_i < T_m} C_i/D_i$ is the utilization of lower priority task invocations that are current at time $t$. Since lower priority invocations do not affect the schedulability of $T_m$, $U(t)$ is minimized when $\sum_{i:T_i < T_m} C_i/D_i = 0$. In other words, one can always reduce the utilization of a critically-schedulable task pattern (in which task invocation $T_m$ has zero slack) by removing all task invocations of priority lower than $T_m$. Thus, to arrive at a minimum utilization bound, $T_m$ must be the lowest-priority task invocation of all those that are current in the interval $A_m \leq t < A_m + D_m$. We call this Property 1. In the following, we shall denote the lowest-priority task invocation by $T_n$, i.e., set $m = n$.

Let $B$ be the end of the last schedule gap (i.e., period of processor idle time) that precedes the arrival of the critically-schedulable task invocation, $T_n$ in some task invocation set $\zeta$. Let $U^\zeta_{\text{max}} = \max_{B \leq t < A_n + D_n} U(t)$. Next, we show that the bound $U_{\text{min}} = \min_{\zeta} U^\zeta_{\text{max}}$ occurs for an invocation set $\zeta$ in which the utilization $U(t)$ is constant in the interval $B \leq t < A_n + D_n$. This interval is shown in Figure 4.

Let $\zeta^*$ be a pattern with the lowest maximum utilization among all critically-schedulable patterns, i.e., $U^\zeta_{\text{max}} = U_{\text{min}}$. By contradiction, assume that $U^\zeta^*(t)$ is not constant in the interval $B \leq t < A_n + D_n$.

\(^4\)Since $U^\zeta$ is also a function of deadlines it can be less than one in the busy interval.
In this case, we show that we can find another critically schedulable invocation pattern with a lower maximum utilization. To do so, choose $t_{hi}$ such that $U^*(t_{hi}) = \max_{B \leq t < A_n + D_n} U^*(t)$. Choose $t_{lo}$ such that $U^*(t_{lo}) < U^*(t_{hi})$. Since $U^*(t_{lo}) < U^*(t_{hi})$, it must be that there exists at least one acyclic task in the pattern such that its current invocation at $t = t_{hi}$, say $T_{hi}$, has a larger utilization than its current invocation at $t = t_{lo}$, say $T_{lo}$, i.e., $C_{hi}/D_{hi} > C_{lo}/D_{lo}$. Consider invocations $T_{hi}$ and $T_{lo}$ of this task. Let us say that invocation $T_A$ delays invocation $T_B$ if a reduction in the completion time of the former increases the slack of the latter and/or vice versa. The following cases arise depending on whether $T_{hi}$ and $T_{lo}$ delay the lowest priority invocation, $T_n$ (which has zero slack):

- **Case 1.** $T_{hi}$ does not delay $T_n$: In this case, reduce the execution time of $T_{hi}$ by an arbitrarily small amount $\delta$. This will decrease the utilization at $t_{hi}$ without affecting the slack of $T_n$.

- **Case 2.** $T_{hi}$ delays $T_n$; $T_{lo}$ delays $T_n$: Reduce the execution time of $T_{hi}$ by an arbitrarily small amount $\delta$, and add $\delta$ to the execution time of $T_{lo}$. The transformation does not decrease the total time that $T_n$ is delayed. However, it has a lower utilization at time $t_{hi}$.

- **Case 3.** $T_{hi}$ delays $T_n$; $T_{lo}$ does not delay $T_n$: Since $T_{hi}$ delays $T_n$ it must be of higher priority. In the case of deadline monotonic scheduling this implies that $D_{hi} < D_n$. Hence, reduce $C_{hi}$ by an arbitrarily small amount $\delta$ and add $\delta$ to $C_n$. This decreases $U(t_{hi})$ by at least $\delta/D_{hi} - \delta/D_n$, where $\delta/D_{hi} > \delta/D_n$ because $D_{hi} < D_n$.

In each transformation above, the slack of $T_n$ is unchanged. The result is a critically schedulable task set of lower maximum utilization. This is a contradiction with the statement that $\zeta^*$ has the lowest maximum utilization among all patterns (i.e., that $U^*(t_{hi}) = U_{\text{min}}$). Hence, the assumption (that $U(t)$ is not constant) is wrong. This proves that $U_{\text{min}}$ occurs when $U(t)$ is constant in the interval $B \leq t < A_n + D_n$. We call this Property 2. Let us call this constant value, $U^\zeta$, where $\zeta$ refers to the task pattern for which this utilization is computed.

We now proceed with minimizing the utilization $U^\zeta$ with respect to the attributes of all task invocations in $\zeta$ that preempt $T_n$. Consider the busy period shown in Figure 4 (i.e., period of continuous CPU
execution) that precedes the deadline of the critically-schedulable task invocation \( T_n \). Let \( L = A_n - B \) be the offset of the arrival time of task invocation \( T_n \) relative to the start of the busy period. The minimization of \( U^i \) with respect to the parameters of the task invocations which execute in this busy period undergoes the following steps:

**Step 1, minimizing \( U^i \) w.r.t. \( L \):** For each acyclic task \( i, 1 \leq i \leq n \), consider the invocation \( T_i \) that arrives last within the busy period \( B \leq t < A_n + D_n \). Let the utilization of this invocation be \( U_i = C_i / D_i \). Since \( T_i \) is the last invocation, its deadline by definition is no earlier than \( A_n + D_n \). Let the sum of execution times of all invocations in \([B, A_n + D_n]\) that precede \( T_i \) in acyclic task \( i \) be \( C_P \). Since, by Property 2, the utilization is constant in the interval \( B \leq t < A_n + D_n \), it must be that \( \sum_{1 \leq i \leq n} C_P = \sum_{1 \leq i \leq n} (A_i - A_n + L) C_i / D_i \). Since the length of the interval is \( D_n + L \), the sum of the execution times within that interval must amount to \( D_n + L \), i.e.:

\[
C_n + \sum_{i=1}^{n} (A_i - A_n + L) \frac{C_i}{D_i} + \sum_{i=1}^{n-1} (C_i - v_i) = D_n + L
\]  

(2)

Where \( v_i \) (which stands for overflow) is the amount of computation time of task invocation \( T_i \) that occurs after the deadline of task invocation \( T_n \). Let \( v = \sum_{1 \leq i \leq n-1} v_i \). Equation (2) can be rewritten as:

\[
C_n = D_n + L(1 - U^i) - \sum_{i=1}^{n-1} (A_i - A_n) \frac{C_i}{D_i} - \sum_{i=1}^{n-1} C_i + v
\]  

(3)

From Equation (3), \( C_n \) (and hence the utilization) is minimum when \( L = 0 \). This implies that in the worst-case invocation pattern, \( T_n \) arrives at the end of an idle period, i.e., \( B = A_n \) (see Figure 4). We call this Property 3. Substituting for \( C_n \) in \( U^i = \sum_{1 \leq i \leq n} C_i / D_i \), then setting \( L = 0 \), the minimum utilization is given by:

\[
U^i = \sum_{i=1}^{n-1} \frac{C_i}{D_i} + 1 - \frac{1}{D_n} \sum_{i=1}^{n-1} (A_i - A_n) \frac{C_i}{D_i} - \sum_{i=1}^{n-1} \frac{C_i}{D_n} + \frac{v}{D_n}
\]  

(4)

**Step 2, minimizing \( U^i \) w.r.t. \( C_i \):** Consider the busy period \( B \leq t < A_n + D_n \), where \( L = 0 \) (i.e., \( B = A_n \)). By Property 1, during this period, \( T_n \) is the lowest priority invocation. Let us minimize \( U^i \) with respect to the computation times of task invocations \( \{T_1, ..., T_{n-1}\} \) where \( T_k \) is the last invocation of the acyclic task \( k \) in this busy period. Let the latest completion time of an invocation in this set be \( E_{last} \). Let \( S_k \) be the start time of some invocation \( T_k \). We consider only invocations, \( T_k \), with non-zero execution times since zero-execution-time invocations do not alter utilization. To minimize \( U^i \) with respect to the computation times of these task invocations, we shall inspect the derivative \( dU^i / dA_k \).
Three cases arise:

1. \( T_k \) arrives while a task invocation of higher priority is running: In this case, \( T_k \) is blocked upon arrival. Advancing the arrival time \( A_k \) by an arbitrarily small amount does not change its start time (and therefore does not change the start or finish time of any other task invocation). Consequently, \( v \) remains constant, and \( dv/dA_k = 0 \). Thus, from Equation (4), \( dU^\zeta/dA_k = -\frac{1}{D_n}(\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\zeta \) can be decreased by increasing the arrival time \( A_k \).

2. \( T_k \) arrives while a task invocation of lower priority is running: In this case \( T_k \) preempts the executing task invocation upon arrival. Advancing the arrival time \( A_k \) by an arbitrarily small amount reorders execution fragments of the two invocations without changing their combined completion time. Consequently, \( v \) remains constant, and \( dv/dA_k = 0 \). Thus, from Equation (4), \( dU^\zeta/dA_k = -\frac{1}{D_n}(\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\zeta \) can be decreased by increasing the arrival time \( A_k \).

3. \( T_k \) arrives while no task invocation is running: In other words, it arrives at or after the completion time of the previously running task invocation. Let us define a contiguous period as a period of contiguous CPU execution of invocations \( T_1, \ldots, T_{n-1} \). The execution of these invocations forms one or more such contiguous periods. Two cases arise:

- A. \( T_k \) is not in the contiguous period that ends at \( E_{last} \): Advancing \( A_k \) will not change \( v \). Thus, \( dv/dA_k = 0 \), and \( dU^\zeta/dA_k = -\frac{1}{D_n}(\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\zeta \) can be decreased by increasing the arrival time \( A_k \).

- B. \( T_k \) is in the contiguous period that ends at \( E_{last} \). Three cases arise: (I) \( E_{last} > D_n \): In this case, \( v > 0 \). Since no other task invocations were running when \( T_k \) arrived, advancing the arrival time of \( T_k \) will shift the last contiguous period and increase \( v \) by the same amount. It follows that \( dv/dA_k = 1 \). Thus, from Equation (4), \( dU^\zeta/dA_k = \frac{1}{D_n}(1 - \frac{C_k}{D_k}) \). This quantity is positive indicating that \( U^\zeta \) can be decreased by decreasing \( A_k \). (II) \( E_{last} < D_n \): In this case, \( v = 0 \). \( dU^\zeta/dA_k = \frac{1}{D_n}(\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\zeta \) can be decreased by increasing the arrival time \( A_k \). (III) \( E_{last} = D_n \): From (I) and (II) above, it can be seen that \( \lim_{E_{last}\rightarrow D_n} dU^\zeta/dA_k \neq \lim_{E_{last}\rightarrow D_n} dU^\zeta/dA_n \). Thus, the derivative \( dU^\zeta/dA_k \) is not defined at \( E_{last} = D_n \). From the signs of the derivative in (I) and (II), it can be seen that \( U^\zeta \) has a minimum at \( E_{last} = D_n \).
From the above, \( U^{\zeta} \) can be decreased in all cases except case 3.B.(III) where a minimum occurs. Since the above cases exhaust all possibilities and 3.B.(III) is the only minimum, it must be a global minimum. In this case, each task invocation \( T_k \) arrives while no task invocation is running (by definition of case 3) and contributes to a contiguous period that ends at \( E_{last} \) (by definition of subcase B) where \( E_{last} = D_n \) (by definition of subcase III). In other words, each task invocation arrives exactly at the completion time of the previous task invocation (for the period to be contiguous) with the completion of the last invocation being \( D_n \). For simplicity, let us re-number invocations \( T_1, ..., T_{n-1} \) in order of their arrival times. It follows that \( U^{\zeta} \) is minimized when \( C_i = A_{i+1} - A_i, \ 1 \leq i \leq n - 2 \) and \( C_{n-1} = D_n - A_{n-1} \) as depicted in Figure 5.

![Figure 5. The Contiguous Period](image)

Let \( T = A_1 - A_n \). Since \( v = 0 \) at the global minimum, from Equation (3), \( C_n = D_n - \sum_{1 \leq i \leq n-1} \{ C_i + (A_i - A_n)C_i/D_i \} \), where \( A_i - A_n = T + \sum_{1 \leq j \leq i-1} C_j \). Substituting for \( A_i - A_n \), we get the expression:

\[
C_n = T(1 - \sum_{i=1}^{n-1} \frac{C_i}{D_i}) - \sum_{i=2}^{n-1} \left( \frac{C_i}{D_i} \sum_{j=1}^{i-1} C_j \right)
\]  

Substituting in the expression \( U^{\zeta} = \sum_i C_i/D_i \), we get:

\[
U^{\zeta} = \frac{T}{D_n} + \left( 1 - \frac{T}{D_n} \right) \sum_{i=1}^{n-1} \frac{C_i}{D_i} - \sum_{i=2}^{n-1} \left( \frac{C_i}{D_i} \sum_{j=1}^{i-1} C_j \right)
\]  

Step 3, minimizing \( U^{\zeta} \) w.r.t. \( D_i \): The utilization in Equation (6) decreases when \( D_1, ..., D_{n-1} \) increase. Since \( T_n \) is the lowest priority invocation, the deadline monotonic scheduling policy upper-bounds these deadlines by \( D_n \). Hence, in a worst-case pattern, \( D_1 = D_2 = ... = D_{n-1} = D_n \). Equation (6) can be significantly simplified in this case. The resulting utilization is given by:

\[
U^{\zeta} = 1 - \frac{T \sum_{i=1}^{n-1} C_i + \sum_{i=2}^{n-1} (C_i \sum_{j=1}^{i-1} C_j)}{(T + \sum_{i=1}^{n-1} C_i)^2}
\]  

15
The corresponding computation time of $T_n$ is given by:

$$C_n = T - \frac{T \sum_{i=1}^{n-1} C_i + \sum_{i=2}^{n-1} (C_i \sum_{j=1}^{i-1} C_j)}{T + \sum_{i=1}^{n-1} C_i} \quad (8)$$

**Step 4, minimizing $U^\zeta$ w.r.t. $T$:** Since arrival times of invocations $T_1, \ldots, T_{n-1}$ are spaced by their respective computation times, as found in Step 2, to obtain the condition for minimum utilization, it is enough to minimize $U^\zeta$ with respect to $T$ subject to the constraint $C_n > 0$. To do so, we first set $dU^\zeta/dT = 0$. Setting the derivative of Equation (7) to zero, we get

$$T = \frac{\sum_{i=1}^{n-1} C_i^2}{\sum_{i=1}^{n-1} C_i^2 / \sum_{i=1}^{n-1} C_i}$$

Substituting this value in Equation (7) and simplifying, we get:

$$U^\zeta = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{\sum_{i=1}^{n-1} C_i^2 / \sum_{i=1}^{n-1} C_i^2} \quad (9)$$

The quantity $\frac{(\sum_{i=1}^{n-1} C_i^2)}{\sum_{i=1}^{n-1} C_i^2}$ in the denominator in Equation (9) is upper bounded by $n - 1$ which corresponds to the case where the computation times are equal, i.e., when $C_i = C, 1 \leq i \leq n - 1$. Consequently, the lower bound on the utilization of a critically schedulable acyclic task set is $U^\zeta = \frac{1}{2} + \frac{1}{2n}$. From Equation (8), we see that under these conditions $C_n = C(1 - \frac{n-1}{2})$. Thus, for $C_n$ to be positive, $n < 3$. Consequently:

$$U^\zeta = \frac{1}{2} + \frac{1}{2n}, \quad n < 3. \quad (10)$$

When $n \geq 3$, the aforementioned bound does not satisfy the constraint $C_n > 0$. Instead, Figure 6 illustrates the minimum utilization condition. The figure plots the relation between $T$ and $U^\zeta$ given by Equation (7), illustrating the point of minimum utilization as well as the constraint $C_n > 0$. Since $C_n$ increases monotonically with $T$, it is obvious from Figure 6 that the point of minimum utilization subject to the constraint $C_n > 0$ is obtained at the value of $T$ that makes $C_n = 0$. Setting $C_n = 0$ in Equation (8) we get $T = \sqrt{\sum_{i=2}^{n-1} (C_i \sum_{j=1}^{i-1} C_j)}$. Substituting in Equation (7), we eventually get:

$$U^\zeta = \frac{1}{1 + \sqrt{\frac{1}{2} \left(1 - \frac{\sum_{i=1}^{n-1} C_i^2}{\sum_{i-1}^{n-1} C_i^2}ight)}} \quad (11)$$

As before, the minimum utilization is obtained when computation times are equal, in which case:

$$U^\zeta = \frac{1}{1 + \sqrt{\frac{1}{2} \left(1 - \frac{1}{n-1}\right)}}, \quad n \geq 3. \quad (12)$$
From Equation (10) and Equation (12) the theorem is proved.

The presented bound is tight in the sense that the utilization of an actual critically-schedulable task pattern can be arbitrarily close to the bound. In the following, we describe a critically schedulable task pattern whose utilization approaches infinitesimally the computed bound for a large $n$. Let $T_n$ have an infinitesimally small execution time. In this example, $T_n$ is blocked throughout the entire interval between its arrival time and deadline by invocations of $n-1$ acyclic tasks. Each task consists of two invocations of equal utilization. The first of these arrives together with $T_n$, say at $t = 0$. The second has an arrival time $\left( j - 1 + \frac{\sqrt{(n-1)(n-2)}}{2} \right) C$, a computation time $C$, and a relative deadline $D = \left( n - 1 + \frac{\sqrt{(n-1)(n-2)}}{2} \right) C$. This implies that the first invocation in task $j$ has a computation time $\left( j - 1 + \frac{\sqrt{(n-1)(n-2)}}{2} \right) C$ and a relative deadline $\left( j - 1 + \frac{\sqrt{(n-1)(n-2)}}{2} \right) C$. This worst case task pattern is shown in Figure 7. Using simple algebra it is easy to see that the utilization of this pattern is equal to the bound and that $T_n$ has zero slack. Hence, the bound is tight.

**Theorem 2:** Deadline monotonic scheduling is an optimal time-independent scheduling policy in the sense of maximizing the utilization bound.

**Proof:** Let $U_x$ be the utilization bound of scheduling policy $X$. Let $U_{dm}$ be the utilization bound for deadline monotonic scheduling, derived above. Consider some utilization value $U$, where $U > U_{dm}$. We shall first show that $U_x < U$.

To show that $U$ is above the schedulability bound of policy $X$ it is enough to demonstrate a single task set of maximum utilization $U$ that is unschedulable by policy $X$. Let us begin by constructing a task invocation pattern (whose maximum utilization is $U$) that is unschedulable by deadline monotonic. This
should always be possible since $U > U_{dm}$. Let the task invocation that misses its deadline in this pattern be called $T_A$. Due to Property 1, without loss of generality, let $T_A$ be the lowest priority invocation under deadline monotonic scheduling. Unless policy $X$ is equivalent to deadline monotonic scheduling, let there be some task invocation $T_B \neq T_A$, such that $T_B$ is the lowest priority task invocation under policy $X$. Since invocation priorities are independent of invocation arrival times, without loss of generality, let $T_B$ be an invocation that arrives together with $T_A$. Let us schedule this task set by policy $X$. The completion of the busy period will remain the same. However, the last invocation to finish is now $T_B$ instead of $T_A$. Since $T_B$ has a higher priority than $T_A$ using deadline monotonic scheduling, it must be that it has a shorter deadline. Hence, task invocation $T_B$ will be unschedulable under policy $X$. Consequently, it must be that $U_x < U$. Taking the limit, $\lim_{U \to U_{dm}}$, it follows that $U_x \leq U_{dm}$ for any scheduling policy $X$.

Corollary 1: The optimal utilization bound of time-independent scheduling of aperiodic tasks is

$$\frac{1}{1+\sqrt{1/2}}$$

This corollary follows directly from Theorem 1 and Theorem 2, combined with the fact that the asymptotic utilization bound for aperiodic tasks is equal to that of acyclic tasks as shown in Section 2.

Corollary 2: A set of periodic tasks (with periods larger than deadlines) is schedulable by the deadline monotonic scheduling policy if $U(t) \leq \frac{1}{1+\sqrt{1/2}}$

While we do not intend this paper to be an analysis of periodic tasks, we notice that the optimal utilization bound derived for aperiodic tasks under deadline-monotonic scheduling also applies to deadline-monotonic scheduling of periodic tasks with deadlines less than periods.\(^5\) This is true for the following two reasons:

- Periodicity is a constraint on the set of possible invocation arrival patterns. This constraint reduces the number of arrival patterns that are permitted but does not offer any new ways a task set can become unschedulable. Thus, the schedulable utilization bound for periodic tasks cannot be lower than that for aperiodic tasks.

- For each unschedulable arrival pattern of aperiodic task invocations that make up a busy interval,

\(^5\)We would like to acknowledge Professor Lui Sha for pointing this observation out to the authors.
one can construct an identical pattern of periodic task invocations with the same execution times and deadlines and whose periods are sufficiently large such that only one invocation of each task appears in the busy interval. Since the two patterns are indistinguishable in terms of both utilization and schedulability status, the schedulable utilization bound for periodic tasks cannot be higher than that for aperiodic tasks.

Putting the above two observation together we conclude that the utilization bound for deadline monotonic scheduling of aperiodic tasks is equal to that of periodic tasks in which deadlines are less than periods.

4 Dynamic Priority Scheduling

For completeness, in this section, we consider dynamic-priority scheduling. It is easy to show that EDF is the optimal dynamic priority scheduling algorithm for aperiodic tasks and that the optimal utilization bound for dynamic priority scheduling is 1. To see that, consider a hypothetical processor capable of generalized processor sharing. The processor assigns each task $T_i$ (of execution time $C_i$ and deadline $D_i$) a processor share equal to $C_i/D_i$. All current tasks execute concurrently. Each task will terminate exactly at its deadline. The maximum schedulable utilization of this processor is trivially $U = 1$. One can imagine that such a processor, in effect, schedules current tasks in a round-robin fashion, assigning each an infinitesimally small time slice that is proportional to its share, $C_i/D_i$. We shall show that this round-robin schedule can be converted into an EDF schedule without causing any task to miss its deadline.

Let us scan the round-robin schedule from its beginning to its end. At every time $t$, let us choose among all current tasks the one with the earliest deadline, say $T_j$, and consolidate its round-robin slices scheduled in the interval $[A_j, t]$. Consolidation is effected by shifting these slices towards the task’s arrival time, displacing slices of tasks that have not been consolidated. Since we are merely switching the order of execution of task slices within an interval that ends at $t$, no slice is displaced beyond $t$. Since task $T_j$ was current at time $t$, it must be that $t \leq A_j + D_j$ (the absolute deadline of $T_j$). Thus, no slice is displaced beyond $T_j$’s deadline. However, since $T_j$ is the task with the shortest deadline among all current tasks at time $t$, no slice is displaced beyond its task’s deadline. The resulting schedule after the scan is EDF. Thus, EDF has the same utilization bound as generalized processor sharing. In other words,
it will cause no aperiodic deadline misses as long utilization is less than 1. This result is similar to the case of sporadic tasks.

5 Experimental Evaluation

Perhaps the most important result of this paper is not that a utilization bound exists for aperiodic tasks but that it is highly efficient in maximizing the number of admitted tasks. In fact, as we show in this section, the measured utilization can approach 100% in a highly loaded system even when the synthetic utilization is kept below the bound. To appreciate this somewhat unintuitive observation, consider the schedule shown in Figure 8.

![Figure 8. Synthetic Utilization](image)

The above schedule depicts four acyclic tasks, each composed of multiple invocations delimited by vertical bars in the figure (note that some invocations have a zero execution time). The contribution of each invocation to synthetic utilization is shown in the figure. At the bottom, a timeline is shown demonstrating the idle and busy intervals of the processor. It is easily seen from that timeline that the measured utilization approaches 100%. However, the aggregate synthetic utilization in this example never exceeds 75%. For example, at time $t_0$, only invocations of tasks 3 and 4 contribute to synthetic utilization, totaling 75%. In particular, the current invocation of task 1 is not counted towards synthetic utilization at $t_0$ because its arrival precedes a schedule gap. The luxury of being able to use schedule gaps for resetting synthetic utilization is not available in offline analysis of periodic tasks. Hence, the encouraging results presented in this section regarding the performance of our utilization-based admission control are unique to aperiodic task scheduling. No corresponding result is possible in utilization-based
schedulability analysis of periodic tasks.

5.1 Measured Utilization of Conforming Task Sets

Let a conforming set of aperiodic tasks be one in which the synthetic utilization never exceeds the bound, i.e., is below 58.6%. This property is enforced by an admission controller which rejects all invocations that would cause the bound to be violated. (Since tasks are aperiodic, the word task and invocation are used interchangeably.) How pessimistic is such an admission control policy in terms of the number of tasks admitted compared the number of tasks that could have made their deadlines? To answer this question, we used a simulator which can generate aperiodic tasks with Poisson arrivals, exponentially distributed execution times, and uniformly distributed deadlines. The simulator implements a deadline-monotonic scheduler and an optional admission controller. An aperiodic task is discarded by the simulator as soon as its remaining computation time exceeds the remaining time until its deadline.

When admission control is used, an incoming arrival is let into the system only if it does not cause the synthetic utilization bound to be exceeded. This eliminates deadline misses at the expense of rejecting a fraction of arrivals. It is assumed that the computation time and deadline of the arrived tasks are known upon arrival. For example, in a web server, the name of the requested URL can be used to lookup the length of the requested page and hence estimate the service time of the request. The admission controller resets the synthetic utilization to zero once the processor becomes idle.

Figure 9 shows the measured CPU utilization and the percentage of tasks that make their deadlines (called success ratio) with and without admission control. The horizontal axis shows the generated incoming load which is varied from 50% to 140% of total CPU capacity. This load is defined as the sum of computation times of all arrived tasks over the time interval during which they arrived.

Each point on each curve in the figure is the average of 10 experiments, each carried out for a sufficiently long time (33,000 task arrivals). Hence, the utilization and success ratio at each point are averaged over more than 3 million task arrivals which gives a sufficiently tight confidence interval in the results. We opted for multiple experiments per point (instead of a single long experiment) to guard against bias for a particular random number generator seed. The seed was varied across different experiments, and each experiment was repeated using the same seed with and without admission control to provide a common-ground comparison of the two cases.

Note that without admission control, the complement of the success ratio shown in Figure 9 represents
the fraction of tasks which miss their deadlines. With admission control, on the other hand, no deadline misses were observed. The complement of the success ratio in this case represents the fraction of rejected tasks. An interesting metric for evaluating admission control schemes is efficiency. We define efficiency as the number of tasks admitted using the admission control scheme divided by the number of tasks that meet their deadlines under the same scheduling policy if no admission control is used. An overly pessimistic admission control heuristic will reject more tasks than needed, causing the efficiency to decrease. Figure 10 shows (from the data in Figure 9) the efficiency of our utilization-based admission control and deadline-monotonic scheduling of aperiodic tasks versus input load when the utilization bound is 58.6% as derived in Section 3.4. It can be seen that the efficiency is 100% at low load. The efficiency dips slightly, reaching a minimum when the incoming load would fully utilize the processor, then improves when the processor is overloaded. Surprisingly, at no time does the efficiency dip below 95%, which is a very encouraging result. The trends in Figure 9 and Figure 10 are explained below.

At low loads, as expected, admission control does not alter utilization. All tasks are let into the system and all meet their deadlines whether admission control is used or not. Hence, efficiency is 100%. As load increases, deadlines are missed in the absence of admission control. In comparison, the presence of admission control causes a slightly larger fraction of tasks to be rejected. Hence efficiency decreases. Presumably, task rejection is more tolerable than deadline misses making the slight decrease in efficiency acceptable. The trend continues until the processor is fully loaded, i.e., until incoming load is 100% of

---

6Note that this is different from comparing to an optimal admission control policy.
Interestingly, beyond this point, as the processor gets overloaded, admission control becomes progressively more efficient. In fact, for loads above 113% efficiency exceeds 100%. Thus, admission control rejects fewer tasks than the number that would miss their deadlines otherwise. This behavior occurs even despite the fact that, in the absence of admission control, tasks which miss their deadlines are not allowed to continue. The explanation of this behavior is twofold. First, in the absence of admission control (which performs the utilization-based schedulability test) it is impossible to know a priori whether or not an executing task will meet its deadline. An initially schedulable task can always get preempted by a higher priority one which arrives later causing the deadline of the former to be missed. Some portion of CPU capacity is thus unavoidably wasted on tasks that end up discarded causing an admission-control-based scheme to be more efficient. To verify this conjecture, Figure 11 compares the utilization due to tasks which meet their deadlines only (called effective utilization) in the cases with and without admission control. Indeed, the effective utilization with admission control exceeds that without admission control when load increases above 123%.

However, since Figure 10 indicates that admission control becomes more efficient at loads above 113% (i.e., at loads even smaller than 123%), a second factor must be affecting efficiency. This factor is that when the processor is overloaded, admission control favors lighter tasks (which would have a higher probability of being admitted). This bias tends to increase throughput in a similar manner as a shortest-job-first policy does, and hence increases success ratio for the same utilization. This conjecture
was verified by plotting the ratio of the average admitted task size (where task size is defined as \( C_i/D_i \)) to the average incoming task size for those experiments shown in Figure 9 in which admission control was used. The result is shown in Figure 12. As expected, this ratio declines as load increases, i.e., admission control favors smaller tasks. With the increase in the number of arrivals, admission control is increasingly more successful at finding smaller tasks. Hence, the effect becomes more pronounced.

The net result is that not only does utilization-based admission control eliminate deadline misses, but also it increases the number of successful tasks. While the particular numbers presented in the figures above would change depending on the load generation process, the qualitative trends and the factors affecting them hold true in general, and indicate very good performance of utilization-based admission control of aperiodic tasks, even under fixed priority scheduling. Results for EDF will obviously be better.

In the following, we conduct a more detailed simulation study evaluating both deadline monotonic and EDF bounds, and comparing the performance to other policies.

The primary variables for the simulations are the inter-arrival distribution and the computation time distribution. The average processor utilization is the ratio of the arrival rate to service rate. Relative deadlines are chosen randomly from a discrete uniform distribution in which the maximum deadline is 9 times the minimum deadline. The simulation was carried out for a duration of 100 times the average deadline value.

5.2 Comparison with Perfect Admission Control

Perfect admission control is defined as an admission control scheme in which we always admit a task if and only if it can be scheduled. Such a control policy can be implemented as follows. Whenever a task arrives, the delay for the newly arrived task is calculated by summing up the computation times of the current higher priority tasks. If the delay plus the computation time requirement of the new task is less than the task’s deadline, the new task is schedulable. The algorithm then checks that the tasks behind the new task in the priority queue remain schedulable as well. If so, the new task is admitted. The perfect admission control algorithm can be implemented in linear time. It serves as a good benchmark to compare to our admission control approach. Perfect admission control would provide the maximum achievable performance if we disregard the implementation overhead.

Figures 13 and 14 compare the aperiodic bound based admission control against the perfect admission control for exponential inter-arrival and computation times. The scheduling policy used is Deadline
Monotonic and EDF respectively. One could conclude that utilization-based admission control compares well with perfect admission control. We lose no more than 5% of schedulability compared to the perfect (globally optimal) algorithm. From Figure 14 observe that, unlike the case with periodic tasks, admission control based on the EDF bound is not globally optimal. This is because some schedulable arrival patterns may cause synthetic utilization to temporarily exceed unity.

An important consideration concerns the implementation overhead of the admission control scheme. Table 1 shows the time taken by simulations (in secs) carried out for a length of 10,000,000 time units. The load was 100% and the average granularities (defined as the ratio of average computation time to average deadline of the tasks) were as shown in the table. When the average granularity of the workload is small, the queue sizes are long and the perfect admission control scheme takes a long time as the algorithm is linear in the number of tasks. For the utilization-based approach, the impact of the increase in the number of tasks is not as severe. The increase in the duration of the simulation in both cases is due to the increase in the total number of tasks that arrive. The increase is higher for perfect admission control because of its higher complexity, lending credibility to the claim that the synthetic utilization-based approach is desirable. The better performance of the synthetic utilization-based approach also
suggests its suitability to networks and high performance web servers where the delays are primarily in
the queue and the granularities are very small; precisely the case in which the advantage of the approach
is observed.

5.3 Granularity of Tasks

Granularity of tasks refers to the size of a task (its computation requirement) as compared to its deadline.
In this experiment, granularity is formally defined as the ratio of average computation time to average
deadline of the workload.

It is interesting to study the real utilization of a resource in the presence of our admission control
heuristic as the granularity of tasks is varied. Figures 15 and 16 show the effect of increasing granularity
for Deadline Monotonic and EDF respectively. One can see that our admission control scheme performs
better at smaller task granularities. This is because at smaller granularities it is less likely to admit
individual tasks with high synthetic utilization that make many other tasks inadmissible. This result
is encouraging, since our target is high performance servers that can process thousands of requests
concurrently (i.e., where individual task granularity is very small).

5.4 Aperiodic and Periodic Tasks

Since periodic tasks can be broken down into aperiodic tasks, the aperiodic utilization bound is applica-
cible to the case of mixed workloads with periodic and aperiodic tasks. In this section, we consider
periodic tasks whose deadline is equal to their period. Observe that since at most one such invocation
can be present at any given time from each periodic task, the synthetic utilization attributed to periodic
tasks is upper bounded by $\sum_{i \in \text{periodic}} C_i / D_i$, which is their utilization factor. Hence, it is possible to simply “allocate” that much synthetic utilization to periodic tasks and subtract it from the utilization bound for aperiodics. Let us call the difference $U_{\text{diff}}$. Observe that the above allocation ensures that periodic tasks always meet their deadlines.

The admission controller keeps track of the synthetic utilization of aperiodic tasks alone. As before the synthetic utilization counter is reset to zero when the processor is idle (i.e., is not running any periodic or aperiodic tasks). If the counter ever reaches $U_{\text{diff}}$ no further aperiodic tasks can be admitted. This ensures that aperiodic tasks never infringe on the capacity allocation made for periodic ones. Scheduling is done as for the aperiodic case. No separate strategy is required for periodic tasks.

Figures 17 and 18 show the real utilization at the resource when different percentages of the workload are composed of periodic tasks. The distribution used for inter-arrival and computation times is exponential. The input utilization of periodic tasks is varied from 0 to the maximum possible utilization in steps of 10%. The aperiodic workload is the same. In the case of deadline monotonic scheduling, Figure 17 shows that initially real utilization remains high as aperiodic tasks are replaced by periodic tasks. However, as the percentage of periodic tasks increases to values close to the utilization bound, fewer aperiodic tasks can get through and the real utilization approaches the reservation made to periodic tasks. Eventually the entire 58.6% (the value of the bound) is allocated to periodic tasks. No aperiodic tasks are allowed. The real utilization, in this case, is equal to the synthetic utilization bound. In the case of EDF scheduling, Figure 18 shows that the utilization remains close to 100% which is the value
of the EDF bound. We conclude that for workloads with a moderate fraction of periodic tasks both
time-independent and dynamic-priority bounds perform well in terms of keeping the processor utilized.
If the number of periodic tasks is large EDF is recommended.

5.5 Comparison with Real Time Queuing Theory

Real Time Queuing Theory was proposed in [15] to incorporate real-time constraints in queuing system
analysis. In [16], admission control techniques were proposed to use results from real time queuing
theory to control lateness of tasks. The theory enables one to provide probabilistic guarantees on the
lateness of tasks. These techniques share similar concerns as that of synthetic utilization-based admis-
sion control and are suitable for aperiodic systems. Some typical results comparing the two approaches
are presented below.

The admission control scheme based on real time queuing theory is mentioned here in brief for conve-
nience. Details on the scheme can be found in [16]. As of now, approaches based on Real Time Queuing
Theory have been proposed only for the EDF scheduling policy. Hence, the experiments to follow shall
compare the bound for EDF only.

The admission controller ensures that

\[ N_{\text{queue}} \leq \frac{Threshold}{\text{Mean Computation Time}} \]

where, \( N_{\text{queue}} \) is the number of tasks in the queue. According to Real Time Queuing Theory, there is a high probability of all tasks
making their deadlines if the total computation time of the tasks in the system (also, referred to as the
total workload) is less than a threshold. In [16], ways are proposed to choose the threshold such that the
probability of deadline misses is low. The probability is a function of the threshold. Assuming that the
deadline distribution of the task set is known, the optimal threshold is known to be close to the mean
deadline [16]. In our experiments the mean deadline was 10,000ms. However, the mean deadline is a
rough estimate in practice, in the case of aperiodic tasks, since it depends on which tasks will arrive.
It is suggested that thresholds slightly lower than the mean deadline should lead to low probability of
deadline misses of tasks admitted.

Figure 19 shows the useful utilization for the utilization-based approach and the real-time queuing
theory-based approach for different threshold values. Useful utilization is defined as the real processor
utilization due to the tasks that meet their deadlines. For the utilization-based approach, all tasks that
are admitted meet their deadlines, however for the queueing theory-based approach, some tasks can fail
to meet their deadlines. Hence, the metric of importance is the utilization of tasks which meet their

28
deadlines or the useful utilization. Figure 20 shows the percentage of admitted tasks. As expected, the performance of real time queuing theory is dependent on the threshold and for thresholds lower than the mean deadline (in this workload, the mean deadline is 10000), the useful utilization is close to 100%.

It can be observed that for the thresholds for which real time queuing theory-based admission control does well, the percentage of admitted tasks is close to that for synthetic utilization-based admission control. However, for loads around 100%, real time queuing theory has been observed to perform better than synthetic utilization consistently. This is to be expected as the queuing theory-based approach calculates the optimal threshold and miss probabilities using workload parameters and hence is better informed than the synthetic utilization-based approach. It is also a probabilistic framework that does not provide deterministic guarantees and is therefore more permissive in its admission control. The utilization-based approach is competitive with real-time queuing theory despite the fact that it must deterministically guarantee deadlines and that it does not assume any knowledge of workload parameters.

6 Related Work

The study of utilization bounds for schedulability of real-time tasks has been an active research topic in real-time computing since the publication of the first such bound by Liu and Layland [18], in 1973. Utilization bounds are the most efficient form of schedulability analysis in terms of computational complexity. Unfortunately, to date, this valuable tool was confined only to task models with periodicity constraints. The development of bounds derived for aperiodic tasks will enable applying this efficient analysis to a myriad of new applications in which workloads are random and do not have periodic be-
behavior on short scales.

The basic utilization bound [18] states that a set of $n$ independent periodic tasks on a uniprocessor will meet their deadlines if utilization is kept below $n(2^{1/n} - 1)$, which converges to 69% as $n$ increases. The bound presents a sufficient (but not necessary) schedulability condition. The logic of the original derivation of this result has been fine-tuned in [6]. In [11], it is shown that if the values of task periods form $K$ harmonic chains, where $K < n$, then the bound can be increased to $K(2^{1/K} - 1)$. In [4], the bound is further improved by considering more information about the task set such as the actual values of task periods. In [26] a design-time technique is described for computing the run-time bound when only periods (but not the execution times) are known a priori. The exact task execution times are plugged-in when the tasks arrive. In [9] task admissibility is improved using a polynomial-time admission control algorithm instead of the utilization bound. The exact characterization of the ability of rate-monotonic scheduling to meet deadlines of periodic tasks is presented in [13]. This characterization derives both sufficient and necessary conditions for schedulability of periodic tasks. A fault-tolerance extension was presented in [25], whose authors consider the overhead of failure recovery from a single fault. The authors prove that each task is recoverable under rate-monotonic scheduling (i.e., the backup replica will complete by the original deadline) if the utilization of the original task set is less than 0.5. This is less pessimistic than the trivial bound of $0.69/2$ (which trivially allows for any task to execute twice). A utilization bound for a modified rate-monotonic algorithm which allows deferred deadlines is considered in [28].

The basic utilization-based schedulability test has also been extended to the multiframe periodic task model in which successive invocations of a task alternate among multiple frames with different execution times [22]. Improvements of the basic multiframe schedulability test are proposed in [8].

Utilization bounds for multiprocessors have received recent attention. In [38], the authors consider different task partitioning heuristics among processors and evaluate their impacts on schedulable utilization. The authors of [24] derive the worst case achievable utilization of a critically schedulable task set using rate-monotonic scheduling on a partitioned multiprocessor. The authors of [19] derive a multiprocessor utilization bound for EDF scheduling. Their bound is shown to be $0.5(n + 1)$, where $n$ is the number of processors. A less pessimistic (but more computationally involved) schedulability test is proposed in [12].
The above research efforts generally share in common the assumption that the task set is known \textit{a priori} and that tasks are periodic (or have a minimum interarrival time). The concept of \textit{dynamic} real-time systems [31], pioneered by the Spring kernel project [32, 33], introduced a paradigm shift in task model assumptions. It focused on applications where run-time workload parameters (such as the task set and its characteristics) are unknown until admission control time. The new model resulted in innovative planning-based scheduling algorithms that provide online schedulability analysis and online guarantees for dynamically arriving tasks [34, 40, 39, 30, 27, 23, 10]. Unfortunately, in this task model, utilization bounds could no longer be used, since task arrivals were, in general, aperiodic.

Aperiodic tasks were handled in previous literature in one of two ways. The first approach requires creation of a high-priority periodic server task for servicing aperiodic requests. Examples include the sporadic server [29], the deferrable server [36], and their variations [17]. The approach bounds the total load imposed on the system by aperiodic tasks allowing critical periodic tasks to meet their deadlines. It usually assumes that aperiodic tasks are soft, and attempts to improve their responsiveness rather than guarantee their deadlines. The second approach typically relies on algorithms for joint scheduling of both hard periodic and aperiodic tasks. It uses a polynomial acceptance test upon the arrival of each aperiodic task to determine whether or not it can meet its deadline. Examples include, aperiodic response-time minimization [14], slack maximization [5], slack stealing [37], the reservation-based (RB) algorithm [3], and the guarantee routines introduced most notably in the Spring kernel [33].

Real Time Queuing Theory ([15]) is an approach which was designed to handle aperiodic task systems by incorporating deadline constraints in standard queuing models. In [16], admission control strategies were discussed to control the lateness of tasks in aperiodic systems. As we have seen, although real time queuing theory is an effective strategy, it requires the knowledge of mean deadlines and other workload parameters to calculate the blocking threshold.

The utilization bound described in this paper is the first constant-time test that enables us to efficiently determine the schedulability of aperiodic workloads. A preliminary version of the results presented here appeared in a conference publication [1]. Future research in aperiodic task scheduling may include extensions of the basic bound that use more information about the task set (e.g., knowledge of specific values of deadlines, specific constraints on task arrival patterns, specific bounds on individual execution times), consider multiple resources, blocking, etc, or explore multiprocessor scheduling.
From the perspective of QoS enforcement, feedback-based mechanisms can be used to maintain the utilization within schedulable limits (e.g., using admission control). In a separate effort [20, 2], the authors developed a feedback control architecture for maintaining the utilization below the bound using control-theoretical techniques. The use of feedback control theory to enforce QoS constraints seems to be gaining popularity as a particularly robust approach in the presence of uncertainty. For example, [35] develops a mechanism for feedback control of operating system queues to maintain smooth performance of pipelined applications. In [7] an algorithm is described for feedback control of a Lotus Notes servers. A similar approach is applied in [21] to a web proxy cache. Utilization bounds can serve as load set points in server performance control loops to guarantee meeting individual request deadlines.

7 Conclusions

In this paper, we derived, for the first time, the optimal utilization bound for the schedulability of aperiodic tasks under time-independent scheduling and under dynamic scheduling policies. The bound results in an $O(1)$ admission test of incoming tasks, which is faster than the polynomial tests proposed in earlier literature. We also showed that deadline monotonic scheduling is an optimal policy in the sense of maximizing the schedulable utilization bound. This result may be the first step towards an aperiodic deadline monotonic scheduling theory — an analog of rate monotonic scheduling theory for the case aperiodic tasks. Such a theory may prove to be of significant importance to many real-time applications such as real-time database transactions, online trading servers, and guaranteed-delay packet scheduling algorithms. In such applications aperiodic arrivals have deadline requirements and their schedulability must be maintained. More importantly, by making a distinction between synthetic and measured utilization, we have shown that the bound can be enforced on the former without considerably affecting the latter. Hence, unlike the case with periodic tasks, aperiodic utilization-based admission control does not underutilize the processor.

While we limited this paper to the first fundamental result, our investigation is by no means complete. Extensions of the theory are needed to the case of dependent tasks, multiple resource requirements, precedence and exclusion constraints, non-preemptive execution, and other task dependencies in a multi-resource environment. Finally, to make the results more usable, it is important to investigate methods for aggregate utilization control that would maintain the utilization below the schedulability bound. Statistical properties of the task arrival process can be combined with mathematical analysis of feedback
control loops to derive probabilistic guarantees on meeting task deadlines. This avenue is currently being pursued by the authors.

**Acknowledgments**

The authors would like to thank Shelby Funk, Deji Chen, Sanjoy Baruah, and Al Mok for their insightful comments on earlier manuscripts of this paper which revealed flaws in the original derivation and suggested several improvements. Special thanks goes to Lui Sha for commenting on the manuscript and suggesting the addition of Corollary 2.

**References**


