Schedulability Analysis and Utilization Bounds for Highly Scalable Real-Time Services *

Tarek F. Abdelzaher, Chenyang Lu
Department of Computer Science, University of Virginia
Charlottesville, VA 22903
E-mail: {zaher,chenyang}@cs.virginia.edu

Abstract

The proliferation of high-volume time-critical web services such as online trading calls for a scalable server design that allows meeting individual response-time guarantees of real-time transactions. A main challenge is to honor these guarantees despite unpredictability in incoming server load. The extremely high volume of real-time service requests mandates constant-time scheduling and schedulability analysis algorithms (as opposed to polynomial or logarithmic ones in the number of current requests).

This paper makes two major contributions towards developing an architecture and theoretical foundations for scalable real-time servers operating in dynamic environments. First, we derive a tight utilization bound for schedulability of aperiodic tasks (requests) that allows implementing a constant time schedulability test on the server. We demonstrate that Liu and Layland’s schedulable utilization bound of \( \ln 2 \) does not apply to aperiodic tasks, and prove that an optimal arrival-time independent scheduling policy will meet all aperiodic task deadlines if utilization is maintained below \( \frac{1}{1 + \sqrt{1/2}} \). Second, we show that aperiodic deadline-monotonic scheduling is the optimal arrival-time independent scheduling policy for aperiodic tasks. This result is used to optimally prioritize server requests. Evaluation of a utilization control loop that maintains server utilization below the bound shows that the approach is effective in meeting all individual deadlines in a high performance real-time server.

1 Introduction

The work reported in this paper is motivated by the timing requirements of high-performance real-time services such as on-line 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. Meeting deadline requirements has traditionally relied on proper scheduling and schedulability analysis techniques. The excessive load of a high performance service puts an important limitation on the complexity of scheduling policies and schedulability analysis techniques that may be applied. In the extreme, both real-time scheduling and run-time schedulability analysis have to take place in constant time. Any algorithms of higher complexity (in the number of served requests) may not scale efficiently in a high-performance server.

Since most current high-performance servers run on non-real-time operating systems which do not implement EDF, in this paper we assume that requests are classified into a limited constant number of classes. All requests of the same class are served in FIFO order at the same priority level. We call such a policy, arrival-time independent since priorities are independent of invocation arrival times. Arrival-time independent policies are a superset of fixed-priority policies. We do not analyze fixed-priority policies because the term usually refers to periodic task scheduling policies in which successive invocations of the same task have the same priority. In the context of aperiodic tasks, where each task has only one invocation, these semantics are inappropriate. Note that the implementation overhead of an arrival-time independent scheduling policy depends only on the number of classes, which is constant by design. Hence, this overhead is constant independent of the number of queued requests.

From the perspective of schedulability analysis, the constant-time processing requirement imposes an important limitation. The most widely-used constant-time schedulability analysis technique is one which uses utilization bounds to determine schedulability. In such an analysis, new incoming requests are admitted into the system only if a utilization bound is not exceeded. Unfortunately, all existing utilization bounds assume that incoming tasks are either periodic or sporadic (with minimum interarrival time). More recently, schedulability bounds were derived for a multiframe periodic task model. In contrast, service requests on a high performance server will, in general, arrive in a completely aperiodic fashion. Hence, real-time scheduling theory needs a new schedulability result that would extend utilization-based schedulability conditions to aperiodic tasks. In this paper, this result is derived and evaluated. We prove that the utilization bound for schedulability

---

*The work reported in this paper was supported in part by the National Science Foundation under grants CCR-0093144 and CCR-0098269.
of aperiodic tasks is $\frac{1}{1+\sqrt{1/2}}$ (i.e., 58.6%). Under an optimal arrival-time-independent scheduling policy all admitted aperiodically arriving tasks are schedulable if the utilization remains below the bound at all times. 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).

It should be noted that schedulable utilization bounds are, by construction, pessimistic. Exceeding the bound does not necessarily entail deadline misses. What we provide is a sufficient schedulability condition that has the merit of being computationally efficient. Hence, we prove that (i) all aperiodic task arrival patterns whose utilization remains below $\frac{1}{1+\sqrt{1/2}}$ are schedulable, and that (ii) there exists an unschedulable pattern of aperiodic task arrivals whose utilization is infinitesimally above $\frac{1}{1+\sqrt{1/2}}$. We expect that servers that can tolerate a certain percentage of deadline misses should set their utilization target above the bound. A thorough study of the relation between excess utilization and the probability of deadline misses is an interesting problem that we defer to future work. We do show in the evaluation section of this paper that for Poisson arrivals, deadline misses are observed when the bound is sufficiently exceeded. We also show that the percentage of missed deadlines increases with increasing excess utilization. Since the bound makes no assumptions regarding task arrival patterns, we expect it to be pessimistic for very bursty arrivals. This is analogous to the pessimism of bounds derived for the sporadic task model that should account for a periodic arrival pattern in the worst case. Fortunately, at high load conditions (which we are most interested in), the CPU load is consistently large thus improving the accuracy of the bound. The bound will perform worse for workloads composed of occasional spurts of task arrivals separated by large idle times. We hope that the results in this paper will inspire more research on more precise constant-time schedulability tests more suitable for extremely bursty loads.

Note that many real-time servers are expected to serve a mix of both real-time and non-real-time traffic. Our paper suggests that while the total server utilization can be close to saturation, the server can naturally meet all real-time deadlines using low-overhead arrival-time-independent scheduling as long as real-time load alone does not exceed 58.6% of total capacity. From the perspective of real-time scheduling theory, we hope that the paper will open the door for several follow-ups that improve the bound, e.g., by considering the characteristics of incoming traffic such as its burstiness in determining schedulability. The derived bound can also be used for run-time load control. For example, a feedback control loop can be used to prevent the utilization from exceeding the bound, hence satisfying the timing requirements. In previous work [2], we implemented such a loop in an Apache web server and demonstrated that it can maintain server utilization below an arbitrary overload threshold by changing the quality of served content such that at higher loads utilization is reduced by serving more clients lower quality content. The bound can simply replace the overload threshold in the mentioned architecture to guarantee meeting timing constraints.

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 arrival-time-independent scheduling of acyclic tasks is described in Section 3. Section 4 evaluates the practicality of the derived bound. Section 5 describes related work. Finally, Section 6 presents the conclusions of the paper and avenues for future work.

2 The Acyclic Task Model

We propose the acyclic task model to represent tasks in a real-time system with no periodicity constraints. 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 $i$ 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. Moreover, we require that the deadline of each task invocation be the arrival time of the next invocation of the same task. Figure 1 shows an example of three acyclic tasks. The horizontal timeliness 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.

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

It is easy to see that any set of aperiodic task invocations can be transfered 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 $d_k$ of the first invocation is no later than the arrival time $A_j$ of the next (i.e., $d_k \leq A_j$). 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, 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 $n$. 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 $n \rightarrow \infty$.

3 The Generalized Bound for Ayclic Tasks

Consider the simple model of scheduling independent acyclic tasks on a uniprocessor. At any given instant, $t$, let $S(t)$ be the set of all task invocations that have arrived but whose deadlines have not expired, i.e., $S(t) = \{T_i | A_i \leq t < A_i + D_i\}$. We call them the current invocations. Let $n(t)$ be the number of current invocations at time $t$. In the acyclic task model, $n(t)$ remains constant over time, i.e., $n(t) = n$. The utilization contributed by the current task invocations, called the current utilization, is $U(t) = \sum_{T_i \in S(t)} C_i / D_i$. We shall prove that using an optimal arrival-time independent scheduling policy, all task invocations will meet their deadlines if $\forall t : U(t) \leq UB(n)$, where $UB(n) = \frac{1}{n} + \frac{1}{n^2}$ for $n < 3$ and $UB(n) = \frac{1}{1 + \sqrt{2(1 - \frac{1}{n^2})}}$ for $n \geq 3$. (At $n = 3$ the two expressions are equivalent.) When the number of current invocations, $n$, increases, the bound approaches $\frac{1}{1 + \sqrt{1/2}}$.

3.1 Arrival-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. Fixed-priority scheduling of acyclic tasks can lead to deadline misses even under an arbitrarily low utilization. This is because different task invocations can have arbitrarily different deadlines and execution times. Hence, associating fixed priority levels with acyclic tasks can lead to a situation where the relative deadline of a low priority task invocation is smaller than the execution time of a concurrent high priority task invocation leading to a deadline miss regardless of the current utilization.

Instead, we consider arrival-time-independent priority scheduling policies. We call a scheduling algorithm arrival-time-independent if the priority assigned to a task invocation does not depend on its arrival time. Typically such policy (i) classifies individual task invocations into a finite number of classes, and (ii) associates a fixed priority with each class (i.e., independently of invocation arrival times). Hence, we formally define arrival-time-independent scheduling as follows:

Definition: An arrival-time-independent scheduling algorithm is a function $f(\tau, t) \rightarrow P$, that:
1. maps an infinite set of task invocations $\tau$ whose arrival times are given by vector $t$ into a finite set of values $P$, and
2. satisfies $f(\tau, t) = f(\tau, t')$, for any $t$ and $t'$.

Note that EDF, for example, is not arrival-time-independent by the above definition, since property 2 is violated. Moreover, an infinite set of task invocations $\tau$ that arrive independently will generally have an infinite set of distinct absolute deadlines, and thus (theoretically) an infinite set of distinct priorities. Hence, $P$ is not finite which violates condition 1. When $P$ is small, an arrival-time-independent scheduling policy can be easily implemented on current operating systems that support a fixed (finite) number of priority levels. This, for example, is akin to diff-serv architectures which classify all network traffic into a finite number of classes and give some classes priority over others.

One arrival-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, an aperiodic deadline monotonic scheduling policy. We shall prove that this policy is optimal.

3.2 The Optimal Utilization Bound

To derive an optimal arrival-time-independent scheduling policy it is necessary to define the sense of optimality. The traditional sense of optimality of a scheduling policy is that for any set of task invocations, if the set is schedulable by some arbitrary policy it is schedulable by the optimal policy. Here, schedulability is an off-line property defined as the ability of invocations to meet their deadlines for any allowable combination of invocation arrival times. This definition
is suitable for periodic and sporadic tasks. In the case of acyclic tasks, no a priori knowledge is assumed about task invocation parameters. Hence, schedulability is an online property, defined only for the particular invocation arrival pattern that occurs at run-time. In this case, no arrival-time-independent policy is optimal in the aforementioned sense. To prove this claim, consider a specific example of a task invocation set composed of two current task invocations, $T_1$ and $T_2$, of execution times 2 and 3, and relative deadlines 3 and 4, respectively. A scheduling policy can prioritize these invocations in only one of two ways, (i) $T_1 > T_2$ (class A policy), or (ii) $T_2 > T_1$ (class B policy). Figure 2-a shows that no class A policy is optimal because an arrival pattern exists that makes the set unschedulable under class A while schedulable under class B. Similarly, Figure 2-b shows that no class B policy is optimal because an arrival pattern exists that makes the set unschedulable under class B while schedulable under class A. An optimal policy, by necessity, would have to be a function of invocation arrival times (such as EDF). Hence, in the context of arrival-time-independent scheduling of aperiodic tasks, we redefine the sense of optimality to mean that a scheduling policy is optimal if it maximizes the schedulable utilization bound.

3. Compute the minimum such utilization $U_{min} = \min_{\zeta \in \mathcal{S}} \{U^\zeta_{max}\}$ 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 $U(t) = U^\zeta_{max}$.

$U_{min}$ 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.

Figure 2. Schedulability of Aperiodic Tasks

In this section, we first derive the utilization bound for schedulability of acyclic tasks under the aperiodic deadline monotonic scheduling policy. We then show that no arrival-time-independent policy can achieve a lower bound. Hence, we show that aperiodic deadline monotonic scheduling is optimal in the sense of achieving the maximum bound. Let a critically schedulable task pattern be one in which some task invocation has zero slack. The derivation of the utilization bound for aperiodic deadline monotonic scheduling undergoes the following steps:

1. Consider the set $\mathcal{S}$ of all possible critically schedulable task invocation patterns.
2. Find, for each pattern $\zeta \in \mathcal{S}$, the maximum utilization $U^\zeta_{max} = \max_{t < T_{dead}} U(t)$ that occurs prior to the deadline of a critically schedulable invocation. By definition of $U^\zeta_{max}$, there exists at least one point (prior to the deadline of a critically schedulable invocation) in the pattern where the utilization $U(t) = U^\zeta_{max}$.

3. Compute the minimum such utilization $U_{min} = \min_{\zeta \in \mathcal{S}} \{U^\zeta_{max}\}$ 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 $U(t) = U^\zeta_{max} \geq U_{min}$.

$U_{min}$ 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 $\mathcal{S}$ defined in step (1). Let the 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 $\mathcal{S}$ 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 current 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, $U^\zeta_{max} = U^\zeta$ where $U^\zeta$ is the constant utilization in the aforementioned interval.\(^1\)

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 aperiodic deadline monotonic scheduling policy if $\forall t : U(t) \leq UB(n), UB(n) = \frac{1}{n+1} + \frac{1}{2n}$ for $n < 3$ and $UB(n) = \frac{1}{1+\sqrt{\frac{1}{n}-\frac{1}{n^2}}}$ 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 $t \in T_m$
within that interval, \( U(t) = \frac{C_m}{D_m} + \sum_{T_i > T_m} C_i/D_i + \sum_{T_i < T_m} C_i/D_i \), where \( C_m/D_m \) is the utilization of task invocation \( T_m \), \( \sum_{T_i > T_m} C_i/D_i \) is the utilization of higher priority task invocations that are current at time \( t \), and \( \sum_{T_i < T_m} C_i/D_i \) 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 \( \sum_{T_i < T_m} C_i/D_i = 0 \). In other words, one can always reduce the utilization of a critically schedulable task pattern (in which task \( T_m \) 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 that are current in the interval \( A_m \leq t < A_n + D_m \). 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 execution gap (i.e., period of processor idle time) that precedes the arrival of the critically schedulable task invocation \( T_m \) in some task invocation set \( \zeta \). Let \( U_{\max}^\zeta = \max_B \zeta \leq A_n + D_n \ U(t) \). Next, we show that the bound \( U_{\min} = \min_B U_{\max}^\zeta \) 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 3.

![Figure 3. The Busy Period](image)

Let \( \zeta^* \) be a pattern with the lowest maximum utilization among all critically-schedulable patterns, i.e., \( U_{\max}^{\zeta^*} = U_{\min} \). By contradiction, assume that \( U^{\zeta^*}(t) \) is not constant in the interval \( B \leq t < A_n + D_n \). 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 \( U^{\zeta^*}(t_{hi}) = \max_B \zeta \leq A_n + D_n \ U^{\zeta^*}(t) \). Choose \( t_{lo} \) such that \( U^{\zeta^*}(t_{lo}) < U^{\zeta^*}(t_{hi}) \). Since \( U^{\zeta^*}(t_{lo}) < U^{\zeta^*}(t_{hi}) \), 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, the execution time of \( T_{hi} \) can be increased 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 aperiodic deadline monotonic scheduling this implies that \( \Delta_{hi} < D_n \). 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., \( U^{\zeta^*}(t_{hi}) = U_{\min} \). Hence, the assumption (that \( U(t) \) is not constant) is wrong. This proves that \( U_{\min} \) 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 3 (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 \( U^{\zeta^*} \) with respect to the parameters of the task invocations which execute in this busy period undergoes the following steps:

**Step 1. minimizing \( U^{\zeta^*} \) 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 \( C_i \). Since, by Property 2, the utilization is constant in the interval \( B \leq t < A_n + D_n \), it must be that \( \sum_{1 \leq i \leq n} C_{pi} = \sum_{1 \leq i \leq n} (A_i - A_n + L)C_i/D_i \). 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.:

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

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 (1) can be rewritten as:

\[
C_n = D_n + L(1 - U^{\zeta^*}) - \sum_{i=1}^{n-1} (A_i - A_n)\frac{C_i}{D_i} + \sum_{i=1}^{n-1} C_i + v \quad (2)
\]
From Equation (2), \( C_n \) (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 3). We call this Property 1. Substituting for \( C_n \) in \( U^\text{C} = \sum_{1 \leq i \leq n} C_i / D_i \), then setting \( L = 0 \), the minimum utilization is given by:

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

(3)

Step 2. minimizing \( U^\text{C} \) 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 \( U^\text{C} \) with respect to the computation times of task invocations \( \{T_1, \ldots, T_{n-1}\} \) 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 \( U^\text{C} \) with respect to the computation times of these task invocations, we shall inspect the derivative \( dU^\text{C} / dA_k \). 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 (3), \( dU^\text{C} / dA_k = -\frac{1}{D_n} (\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\text{C} \) 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 (3), \( dU^\text{C} / dA_k = -\frac{1}{D_n} (\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\text{C} \) 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 \( dU^\text{C} / dA_k = -\frac{1}{D_n} (\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\text{C} \) 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 (3), \( dU^\text{C} / dA_k = \frac{1}{D_n} (\frac{C_k}{D_k}) \). This quantity is positive indicating that \( U^\text{C} \) can be decreased by decreasing \( A_k \). (II) \( E_{last} < D_n \): In this case, \( v = 0 \). \( dU^\text{C} / dA_k = -\frac{1}{D_n} (\frac{C_k}{D_k}) \). This quantity is negative indicating that \( U^\text{C} \) 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 \( \lim_{E_{last} \to D_n} dU^\text{C} / dA_k \neq \lim_{E_{last} \to D_n} dU^\text{C} / dA_k \). Thus, the derivative \( dU^\text{C} / dA_k \) is not defined at \( E_{last} = D_n \). From the signs of the derivative in (I) and (II), it can be seen that \( U^\text{C} \) has a minimum at \( E_{last} = D_n \).

From the above, \( U^\text{C} \) 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 \( T_1, \ldots, T_{n-1} \) in order of their arrival times. It follows that \( U^\text{C} \) 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 4.

![Figure 4. The Contiguous Period](image-url)
\[ C_n = T \left( 1 - \sum_{i=1}^{n-1} \frac{C_i}{D_i} \right) - \sum_{i=1}^{n-2} \left( C_i \sum_{j=i+1}^{n-1} \frac{C_j}{D_j} \right) \]  

Substituting in the expression \( U^C = \sum C_i / D_i \), we get:

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

**Step 3**, minimizing \( U^C \) w.r.t. \( D_i \): The utilization in Equation (5) decreases when \( D_1, \ldots, D_{n-1} \) increase. Since \( T_n \) is the lowest priority invocation, the aperiodic deadline monotonic scheduling policy upper-bounds these deadlines by \( D_n \). Hence, in a worst case pattern, \( D_1 = D_2 = \ldots = D_{n-1} = D_n \). Equation (5) can be significantly simplified in this case. The resulting utilization is given by:

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

The corresponding computation time of \( T_n \) is given by:

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

**Step 4**, minimizing \( U^C \) 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 \( U^C \) with respect to \( T \) subject to the constraint \( C_n > 0 \). To do so, we first set \( dU^C / dT = 0 \). Setting the derivative of Equation (6) to zero, we get

\[ T = \sum_{i=1}^{n} C_i / \sum_{i=1}^{n} C_i \]  

Substituting in Equation (6) and simplifying, we get:

\[ U^C = \frac{1}{2} + \frac{1/2}{\left( \sum_{i=1}^{n} C_i \right)^2 / \sum_{i=1}^{n} C_i} \]  

The quantity \( \sum_{i=1}^{n} C_i \) in the denominator in Equation (8) is upper bounded by \( n \), 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

\[ U^C = \frac{1}{2} + \frac{1/2}{2n} \]  

From Equation (9), when \( n \geq 3 \), the aforementioned bound does not satisfy the constraint \( C_n > 0 \). Instead, Figure 5 illustrates the minimum utilization condition. The figure plots the relation between \( T \) and \( U^C \) given by Equation (6), 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 5 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 (7) we get

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

Substituting in Equation (6), we eventually get:

\[ U^C = \frac{1}{2} + \frac{1}{2n}, \quad n \geq 3 \]  

**Theorem 2**: Aperiodic deadline monotonic scheduling is an optimal arrival-time-independent scheduling policy in the sense of minimizing the utilization bound.

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

![Figure 5. The Minimum Utilization Condition](image-url)
The utilization bound of aperiodic deadline monotonic scheduling is optimal among arrival-time-independent scheduling policies.

4 Experimental Evaluation

To check if the utilization bound is accurate, we used a combination of experimental testing of a real server prototype, and simulation results obtained from a home-brewed simulator. These two sets of experiments are presented in the next two subsections respectively.

4.1 Testing a Real Prototype

In previous work we demonstrated the success of a utilization control loop in controlling web server performance. The loop is implemented as a part of our utilization control middleware described in more detail in [2]. This middleware measures average server utilization periodically and manipulates the quality of served content such that the measured utilization is maintained around a desired operating point. In the experiment described in this section, the middleware was configured to maintain server utilization at the bound. The middleware library was linked to an Apache web server on a Linux platform. The combination of Apache over Linux is representative of many web server configurations today. The experimental server platform was an AMD-based PC connected via a local area network to client machines. Several machines were used to run client software that tests the server with a synthetic workload. We used a web-load generator, called httperf [10], on the client machines to bombard the server with web requests. Since the number of concurrent requests was very large, in essence the number of equivalent acyclic tasks $n$ was infinite.

Figure 7 depicts the achieved utilization. In this experiment, the request rate on the server was increased suddenly, at time $= 0$, from zero to a rate that exceeds the schedulable utilization bound. Such a sudden load change approximates a step function. Figure 7 compares the open loop server utilization to the closed loop utilization with controller gain margin values of $G = 4$ and $G = 10$ (where a higher gain margin indicates more conservative utilization control). As shown in Figure 7, the loop was successful in reducing server utilization to remain around the target for the duration of the experiment.

![Figure 7. Utilization Control Performance](image-url)
independent aperiodic tasks with firm deadlines (i.e., tasks were immediately aborted when they missed their deadlines). The simulator was composed of a set of sources that generated tasks, an executor that emulated the scheduling and execution of tasks, and a monitor that periodically collected performance statistics (miss ratio and CPU utilization) of the system. The deadline monotonic policy was used to schedule the tasks. The inter-arrival-time between subsequent tasks from a source $k$ followed an exponential distribution. In our experiments, the execution time $C_k$ of tasks from a source $k$ was randomly generated from a uniform distribution in the range between $[12.5, 50]$ time units. For the purpose of presentation, we assume each time unit is a millisecond. The slack factor of a source $k$, defined as $F_k = D_k/C_k - 1$, followed a uniform distribution in the range $[32, 48]$.

The sampling period was $S = 6$ sec in our experiments. At every sampling instant $t$, the monitor collected the following information during the last sampling period $(t - S, t)$, where $S$ is the constant sampling period:

- **Sampled miss ratio $SM(t)$** is defined as the number of deadline misses divided by the number of tasks that terminate in the sampling period $(t - S, t)$.

- **Sampled CPU utilization $SU(t)$** is defined as the percentage of busy time during the sampling period $(t - S, t)$. Note that the sampled CPU utilization is an approximation to the current utilization of the CPU.

In addition, the monitor computed the maximum sampled miss ratio $SM_{\max}$ defined as the maximum sampled miss ratio in a run. It represents the worst transient miss ratio due to workload burstiness. $SM_{\max} = 0$ means that there is no deadline misses in a run. Finally, the monitor computed the average CPU utilization $AU$ defined as the percentage of busy time during a run.

To measure the tightness of our utilization bound, we repeated the run with different numbers of sources. Each run lasted for $1200$ sec. Figure 8 plots the maximum sampled miss ratio corresponding to each run.

We can see that no deadline misses occurred when the average CPU utilization was below the bound. However, when $AU > 66\%$, the maximum sampled miss ratio was nonzero, $SM_{\max} > 0$, which means that deadline violations occurred under this load condition. This result demonstrates that sampled average CPU utilization can approximate the logical utilization defined in the derivation of the bound. Hence, measured utilization can be used for admission control purposes to avoid deadline misses. The difference between the theoretical bound and the measured threshold of deadline misses is attributed to the task generation process which apparently didn’t generate the worst case pattern.

![Figure 8. Maximum Miss Ratio](image)

**Figure 8. Maximum Miss Ratio**

Figures 9 plots the traces of sampled CPU utilization and miss ratio for four different workload levels. As expected, both the frequency and magnitude of non-zero sampled miss ratio increased as the average CPU utilization increased. When the average CPU utilization was $66.83\%$, deadline misses occurred only sporadically. Deadline misses occurred significantly more frequently when the average CPU utilization was higher. This result justifies the use admission control based on sampled CPU utilization as a mechanism to avoid deadline misses in a system with an aperiodic task workload.

In the second set of simulation experiments, we used the utilization bound as a set point to an adaptive feedback control loop that automatically adjusts task service quality parameters such that a desired utilization is maintained. In a practical application, quality would correspond to frame rate, resolution, color depth, content compression, or other attributes that can be changes by the server to alter CPU load. A quality level of zero was added to represent denial of service. The server was subjected to simulated overload. We repeated the experiments with two different CPU uti-
lization set points; 58.6% (our utilization bound), and 80%. The resulting run-time server utilization and miss ratio for the two cases are depicted in Figure 10-(a) and Figure 10-(b) respectively. In both runs, the service quality control module was initialized to a 0 (no service). This QoS level was subsequently increased by the simulated control loop until the desired utilization was reached. We can see that, when the utilization bound of 58.6% was used as the utilization set point, the system had no deadline misses. In comparison, when the utilization set point was 80%, the system suffered deadline misses in several sampling periods. This result demonstrates the effectiveness of our utilization bound in feedback-based quality adaptation for the purposes of avoiding deadline misses during overload.

We envision a fourth paradigm for real-time scheduling that concerns aperiodic tasks (such as requests on a web server) whose execution times (or more generally, resource requirements) are unknown even after admission. The uncertainty in resource requirements may be due, for example, to data-dependencies that make it impossible to predict the execution time of a task without interpreting the semantics of its application-specific inputs. One measurable quantity in such systems would be the aggregate utilization of the different resources. Theory is needed to relate such utilization to the schedulability of aperiodic tasks. To date, no utilization-based schedulability test has been proposed for the aperiodic task model.

Aperiodic tasks are handled in prior 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 [13], the deferrable server [17], and their variations [9]. 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 [8], slack maximization [7], slack stealing [18], the reservation-based (RB) algorithm [4], and the guarantee routines introduced most notably by the Spring kernel project [16]. Other notable work on aperiodic scheduling includes results on EDF scheduling [5], dynamic scheduling with precedence constraints [6], and robust dynamic preemptive scheduling [3]. In addition to being of higher complexity than utilization-based tests, typical admission control algorithms must know the worst case execution times of arrived tasks.

In contrast, we investigated the problem of deriving a utilization bound for guaranteed schedulability of aperiodic tasks. With the plethora of QoS adaptation mechanisms described in earlier literature, feedback-based QoS-adaptation can be used to maintain the utilization within schedulable limits. Future work of the authors is concerned with developing probabilistic guarantees on maintaining the utilization below the bound using control-theoretical techniques. Of particular interest is the problem of relating measured utilization to the abstract definition of utilization used in this paper.

5 Related Work

To date, three main paradigms have been proposed for real-time scheduling. Perhaps the earliest approach to providing guarantees in performance-critical systems has been to rely on static allocation and scheduling algorithms that assume full a priori knowledge of the resource requirements of tasks and their arrival times [20, 12, 19, 1]. Rate monotonic scheduling theory [11] introduced a second paradigm in which knowledge of task arrival times is not required. As a result, sporadic tasks could be accommodated as long as their minimum inter-arrival time is known. The concept of dynamic real-time systems [14], pioneered by the Spring kernel project [15, 16], introduced the third major paradigm to describe applications where run-time workload parameters are unknown until admission control time.

Figure 10. Feedback-Based QoS Adaptation for Satisfying Deadline Constraints

6 Conclusions

In this paper, we derived, for the first time, the optimal utilization bound for the schedulability of aperiodic tasks under arrival-time-independent scheduling. The bound allows an $O(1)$ admission test of incoming tasks, which is faster
than the polynomial tests proposed in earlier literature. We also showed that aperiodic 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. In such applications aperiodic arrivals have deadline requirements and their schedulability must be maintained.

While we limited this paper to the first fundamental result, our investigation is by no means complete. We will explore in subsequent publications extensions of the theory 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. We shall also extend our results to multiprocessor scheduling of aperiodic tasks. While a multiprocessor can be trivially considered as a set of uniprocessors, it is interesting to investigate whether or not better bounds are possible when all processors share a single run queue.

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 author.

Acknowledgements

The authors would like to thank Shelby Funk and Sanjoy Baruh for their rigorous review and shepherding of an earlier manuscript of this paper which revealed several flaws in the original derivation. The authors would also like to thank the anonymous referees for their valuable comments.

References


