Evaluating the Average-case Performance Penalty of Bandwidth-like Interfaces

Björn Andersson
Carnegie Mellon University

ABSTRACT

Many solutions for composability and compositionality rely on specifying the interface for a component using bandwidth. Some previous works specify period (P) and budget (Q) as an interface for a component. Q/P provides us with a bandwidth (the share of a processor that this component may request); P specifies the time-granularity of the allocation of this processing capacity. Other works add another parameter *deadline* which can help to provide tighter bounds on how this processing capacity is distributed. Yet other works use the parameters α and Δ where α is the bandwidth and Δ specifies how smoothly this bandwidth is distributed. It is known [4] that such bandwidth-like interfaces carry a cost: there are tasksets that could be guaranteed to be schedulable if tasks were scheduled directly on the processor, but with bandwidth-like interfaces, it is impossible to guarantee the taskset to be schedulable. And it is also known that this penalty can be infinite, i.e., the use of bandwidth-like interfaces may require the use of a processor that has a speed that is k times faster, and one can show this for any k. This brings the question: "What is the average-case performance penalty of bandwidth-like interfaces?" This paper addresses this question. We answer the question by randomly generating tasksets and then for each of these tasksets, compute a lower bound on how much faster a processor needs to be when a bandwidth-like scheme is used. We do not consider any specific bandwidth-like scheme; instead, we derive an expression that states a lower bound on how much faster a processor needs to be when a bandwidth-like scheme is used. For the distributions considered in this paper, we find that (i) the experimental results depend on the experimental setup, (ii) this lower bound on the penalty was never larger than 4.0, (iii) for one experimental setup, for each taskset, it was greater than 2.4, (iv) the histogram of this penalty appears to be unimodal, and (v) for implicit-deadline sporadic tasks, this lower bound on the penalty was exactly 1.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Performance, Theory

Keywords

Real-time, Composability, Compositionality

1. INTRODUCTION

Consider a taskset τ scheduled on a single processor with Earliest-Deadline-First (EDF). Assume that tasks are arbitrary-deadline sporadic tasks, i.e., a task τ_i is characterized by T_i, D_i, and C_i, with the interpretation that τ_i generates a sequence of jobs with at least T_i time units between two consecutive arrivals of jobs of τ_i and each job of τ_i has execution time at most C_i and each job has a deadline D_i time units relative to its arrival. It is known [5] that if for all positive t it holds that

$$\sum_{\tau_i \in \tau} \max(\lfloor \frac{t-D_i}{T_i} \rfloor + 1, 0) \times C_i \leq t \quad (1)$$

then the taskset is schedulable. (A taskset is schedulable, if for each jobset that it can generate, for each schedule that EDF can generate for this jobset, it holds that all jobs meet deadlines; note that because we assume EDF with arbitrary tie-breaking, there may be more than one valid schedule for a jobset scheduled by EDF.) The result above (in Eq. 1) is well-known and allows software practitioners to efficiently verify, before run-time, that all timing requirements will be met at run-time. This result works assuming that the entire taskset is known to a single person (or schedulability analysis tool) and that the system does not undergo design changes and that tasks do not use more resources than stated by their parameters.

The real-time systems research community understood these limitations of using Eq. 1 as a schedulability test and using EDF at run-time. The research community understood that it was necessary to monitor the execution of a task to see if it executed more than it was expected to, and also to monitor a task to see if it generated jobs more frequently than it was expected to. In addition, the research community understood that in system integration, it is often advantageous to describe a set of tasks with related functionality with a simpler description. Therefore, the research community created a large set of solutions to achieve this. Typically, these solutions work as follows. A component (sometimes called a

server task and sometimes called a subsystem) is characterized by a bandwidth parameter and some other parameters. One or more tasks are assigned to a component. And each task is assigned to exactly one component. Then, at each instant at run-time, a root scheduler (sometimes called global scheduler) selects a component and a local scheduler in this component selects a task in this component. This selected task executes on the processor. The bandwidth of a component is characterized as a share of the processor (for example, Component 1 should use at most 20% of the processor). One could use a run-time mechanism that guarantees that this bandwidth is allocated to each component in each time interval (even "infinitely small" time intervals). But this would require infinitely many context switches which would make such a solution impractical. Therefore, the solutions presented in the research literature use another parameter as well: server-period. Some solutions ensure that in a time interval of duration at least as large as the server period, a component is allocated processing time being at least as large as its bandwidth (i.e., bandwidth multiplied by server period). Many schemes in the literature suffer from a so-called blackout period; for such a scheme, the guaranteed allocation is slightly less. Some schemes have a parameter, server deadline, which can be used to control the duration of this blackout period. Common to these schemes, however, is that before run-time, a schedulability test is performed on the root scheduler; it takes the bandwidth and potentially other parameters of each component and determines if the root scheduler will be able to allocate enough bandwidth to each component.

These bandwidth-like schemes have several advantages. First, they achieve isolation. This is important because it is often very difficult to find the worst-case execution time of a task. With these schemes, one can be sure that if the execution time of a job would exceed its estimated worst-case execution time, then it does not jeopardize timing guarantees of jobs in other components. Second, they allow hard and soft real-time tasks to be executed on a single processor. Third, they provides a simple interface for system integrators. In particular, the concept of bandwidth is easy to understand for laypersons. (For example: Component 1 is assigned 10% of the processor; Component 2 is assigned 70% of the processor; Component 3 is assigned 15% of the processor.) Fourth, they allow different schedulers to be used in different components (e.g., the local scheduler in Component 1 may be EDF and the local scheduler in Component 2 may be RM and the local scheduler in Component 3 may be FIFO.) Fifth, some real-time operating systems support their run-time mechanisms. Sixth, the run-time policing has low time and space complexity. Given all these advantages, it may seem that bandwidth-like schemes offer a good foundation for real-time systems.

Unfortunately, bandwidth-like schemes can waste an infinite amount of resources. It can be seen as follows: Consider a taskset τ with n tasks, τ_1, τ_2, ..., τ_n and these tasks have the parameters $T_i = \infty$,$D_i = i$,$C_i = 1$. If these tasks are scheduled directly on the processor (i.e., without components), it can be seen that the taskset is EDF-schedulable according to Eq. 1. Let us now discuss their behavior in a system with components and with a bandwidth-like scheme. Suppose that there are n components and one task in each component; specifically, task τ_i is assigned to component i. In order for the root scheduler to be schedulable, it is required that the sum of bandwidth of the components is at most 1. The bandwidth required by a component depends on the actual scheme used (the blackout period matters and the predictability of supply of the root scheduler matters); but for all bandwidth-like schemes, it holds that the required bandwidth of a component is at least as large as the sum of the density of the tasks in the component. This yields that component i requires at least the bandwidth C_i/D_i. Hence, component i requires at least the bandwidth $1/i$. Consequently, in order for the root scheduler to be schedulable, it must hold that $\sum_{i\in\{1..n\}} 1/i \leq 1$. It is easy to see that (for $n \geq 2$) this condition is false and hence the system is not schedulable with a bandwidth-like interface. Let the processor be k times faster. Then, in order for the root scheduler to be schedulable, it must hold that $\sum_{i\in\{1..n\}} 1/i \leq k$. Letting n approach infinity yields that $\sum_{i\in\{1..n\}} 1/i$ approaches $\ln n$ and hence it approaches infinity. Thus, even using a processor that is k times faster cannot guarantee that the system is schedulable with a bandwidth-like interface. We can do this reasoning for any k and hence we obtain that bandwidth-like interfaces can generate an infinite waste of resources.

This observation (that bandwidth-like interfaces can waste an infinite amount of resources) is known in the literature [3, 4]. What is not known, however, is how well bandwidth-like schemes perform in the average-case as compared to a scheme that performs flat scheduling. Exploring this question is the goal of this paper.

2. FORMULATING THE PROBLEM

From Eq. 1 it can be seen that for a taskset scheduled with EDF, a processor speed

$$\max_{t>0}(\sum_{\tau_i\in\tau} \max(\lfloor \frac{t-D_i}{T_i} \rfloor + 1, 0) \times \frac{C_i}{t}) \quad (2)$$

is sufficient to meet deadlines.

From the discussion in the previous section, it can be seen that if each task is in its own component, then with a bandwidth-like scheme, a processor speed

$$\sum_{\tau_i\in\tau} \frac{C_i}{\min(D_i,T_i)} \quad (3)$$

is necessary to meet deadlines.

For a taskset τ, let $\text{spdf}(\tau)$ be defined as:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i\in\tau} \frac{C_i}{\min(D_i,T_i)}}{\max_{t>0}(\sum_{\tau_i\in\tau} \max(\lfloor \frac{t-D_i}{T_i} \rfloor + 1, 0) \times \frac{C_i}{t})} \quad (4)$$

Here $\text{spdf}(\tau)$ should be read as speed-up factor. Intuitively, $\text{spdf}(\tau)$ indicates a lower bound on how much faster the processor needs to be in order for a bandwidth-like scheme to make the taskset τ schedulable.

We have already seen that there is a taskset such that spdf is infinite. This paper explores the question: What is $\text{spdf}(\tau)$ for typical tasksets? The next four sections consider different assumptions and derive expressions for $\text{spdf}(\tau)$. For the first three next sections, we generate tasks randomly and obtain histograms of $\text{spdf}(\tau)$. For the fourth next section, we give a simple analytic expression of $\text{spdf}(\tau)$ and hence do

not run experiments. After these four sections, we comment on the experimental results.

3. EVALUATION FOR THE SPECIAL CASE OF TASKSETS WITH INFINITE MINIMUM INTER-ARRIVAL TIMES

In this section, we consider the special case where for each task τ_i in τ, it holds that $T_i = \infty$.

For this case, $\text{spdf}(\tau)$ can be computed as:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{D_i}}{\max_{t>0}(\sum_{\tau_i \in \tau} \theta(t-D_i) \times \frac{C_i}{t})} \quad (5)$$

where θ is the step function (it returns 1 if its input is non-negative and it returns 0 if its input is negative).

Look at the denominator. Note that this step function only changes for those t that are equal to a D parameter. It can be seen that we only need to check those t that are equal to a D parameter. Using this observation yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{D_i}}{\max_{\tau_j \in \tau}(\sum_{\tau_i \in \tau} \theta(D_j-D_i) \times \frac{C_i}{D_j})} \quad (6)$$

Look at the denominator. Observe that we only need to include the terms where $D_j - D_i \geq 0$; the other ones are zero. With this observation, additional rewriting yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{D_i}}{\max_{\tau_j \in \tau}(\sum_{\tau_i \in \tau \ s.t. \ D_i \leq D_j} \frac{C_i}{D_j})} \quad (7)$$

We will explore $\text{spdf}(\tau)$ for randomly-generated tasksets. We do it with two types of taskset generation: similar tasks and very different tasks.

With the taskset generation *similar tasks*, we generate tasks as follows: $D_i = \text{random}(1, 10)$, $C_i = \text{random}(1, 10)$, where $\text{random}(a, b)$ generates a random number, a real number, in the range [a,b] and it does so with a uniform distribution. If $C_i > D_i$ then we swap the values so that $C_i \leq D_i$. After that, we normalize the density of the taskset so that it is 0.999; i.e., we multiple the C-parameters of all tasks so that the density becomes 0.999. It can be seen that this does not change $\text{spdf}(\tau)$; we do it simply so that we get taskset parameters that are easier to interpret. Then we sort the tasks in ascending order of the D parameter.

With the taskset generation *very different tasks*, we generate tasks as follows: $C_i = 10^{\text{random}(0,4)} \times \text{random}(1, 10)$, $D_i = 10^{\text{random}(0,4)} \times \text{random}(1, 10)$, where $\text{random}(a, b)$ generates a random number, a real number, in the range [a,b] and it does so with a uniform distribution. If $C_i > D_i$ then we swap the values so that $C_i \leq D_i$. After that, we normalize the density of the taskset so that it is 0.999; i.e., we multiple the C-parameters of all tasks so that the density becomes 0.999. Then we sort the tasks in ascending order of the D parameter.

The results for both *similar tasks* and *very different tasks* are shown in Figure 1. Each subplot shows a histogram of 3000 randomly generated tasksets.

4. EVALUATION FOR THE GENERAL CASE, ARBITRARY-DEADLINE SPORADIC TASKS

In this section, we consider the general case, i.e., it is not required that T parameters are infinite. It is also assumed that the taskset has arbitrary deadlines; i.e., D_i is allowed to be less than, or equal to, or greater than T_i.

Recall from Eq. 4 that

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{\min(D_i, T_i)}}{\max_{t>0}(\sum_{\tau_i \in \tau} \max(\lfloor \frac{t-D_i}{T_i} \rfloor + 1, 0) \times \frac{C_i}{t})} \quad (8)$$

Note that the denominator has an expression which considers all positive t. One can see, however, that if only those t for which there exists a task τ_j and a non-negative integer k such that $t = k \times T_j + D_j$ are considered, then the calculation yields the same result. One can also see that if only values of t that are at most $10 \times \max_{\tau_i \in \tau}(T_i + D_i)$ are considered, then the calculated value is an approximation of $\text{spdf}(\tau)$ that has an error of at most 10%. We will use this to obtain an approximation of $\text{spdf}(\tau)$.

We will explore $\text{spdf}(\tau)$ for randomly-generated tasksets. We do it with two types of taskset generation: similar tasks and very different tasks.

With the taskset generation *similar tasks*, we generate tasks as follows: $D_i = \text{random}(1, 10)$, $C_i = \text{random}(1, 10)$, $T_i = \text{random}(1, 10)$, where $\text{random}(a, b)$ generates a random number, a real number, in the range [a,b] and it does so with a uniform distribution. If $C_i > D_i$ then we swap the values so that $C_i \leq D_i$. After that, we normalize the utilization of the taskset so that it is 0.999; i.e., we multiple the C-parameters of all tasks so that the utilization becomes 0.999. Then we sort the tasks in ascending order of the D parameter.

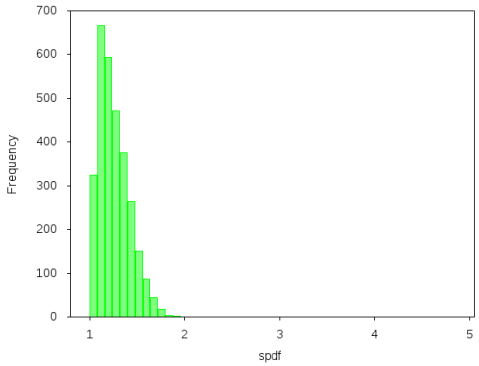
With the taskset generation *very different tasks*, we generate tasks as follows: $C_i = 10^{\text{random}(0,4)} \times \text{random}(1, 10)$, $D_i = 10^{\text{random}(0,4)} \times \text{random}(1, 10)$, and $T_i = 10^{\text{random}(0,4)} \times \text{random}(1, 10)$, where $\text{random}(a, b)$ generates a random number, a real number, in the range [a,b] and it does so with a uniform distribution. If $C_i > D_i$ then we swap the values so that $C_i \leq D_i$. After that, we normalize utilization as mentioned above. Then we sort the tasks in ascending order of the D parameter.

The results for both *similar tasks* and *very different tasks* are shown in Figure 2. Each subplot shows a histogram of 3000 randomly generated tasksets.

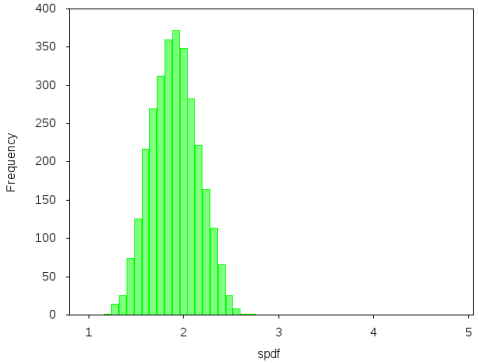
5. EVALUATION FOR THE GENERAL CASE, CONSTRAINED-DEADLINE SPORADIC TASKS

In this section, we consider the general case, i.e., it is not required that T parameters are infinite. It is also assumed that the taskset has constrained deadlines; i.e., D_i is allowed to be less than or equal to T_i.

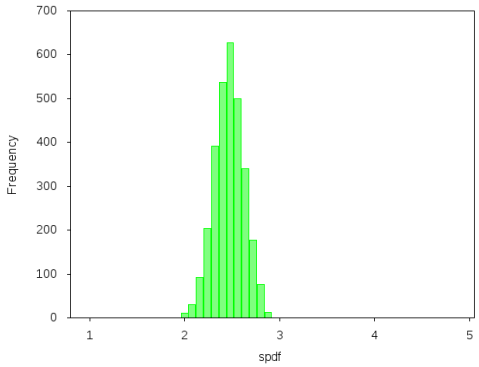
The results for both *similar tasks* and *very different tasks* are shown in Figure 3. Each subplot shows a histogram of 3000 randomly generated tasksets.



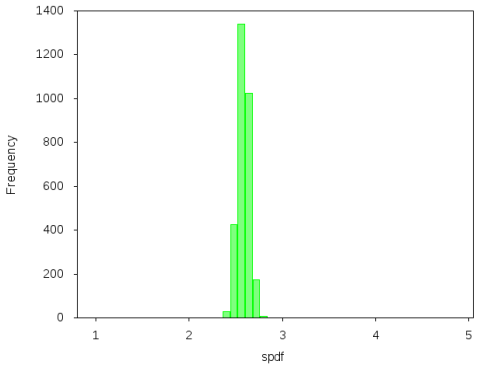
(a) Tasks similar and $|\tau|$=2.



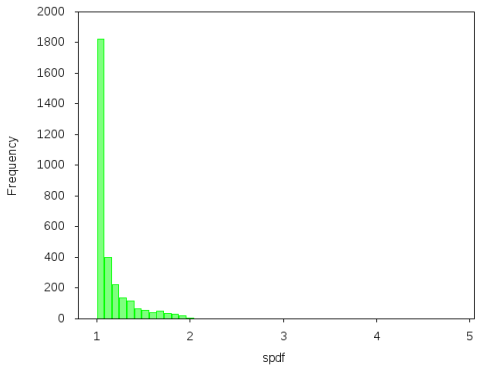
(b) Tasks similar and $|\tau|$=10.



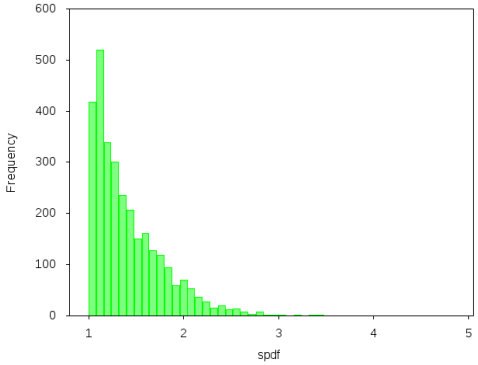
(c) Tasks similar and $|\tau|$=100.



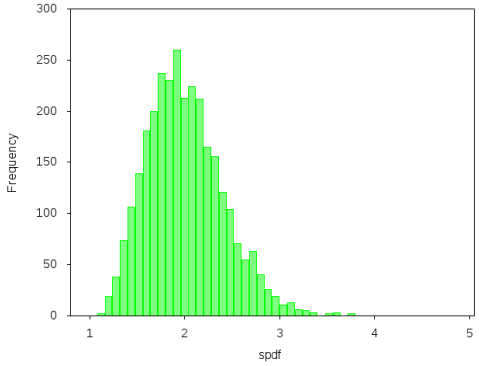
(d) Tasks similar and $|\tau|$=1000.



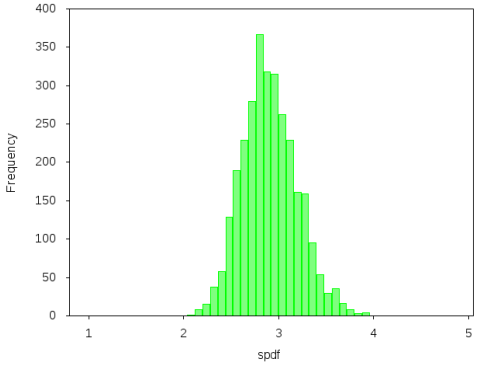
(e) Tasks not similar and $|\tau|$=2.



(f) Tasks not similar and $|\tau|$=10.

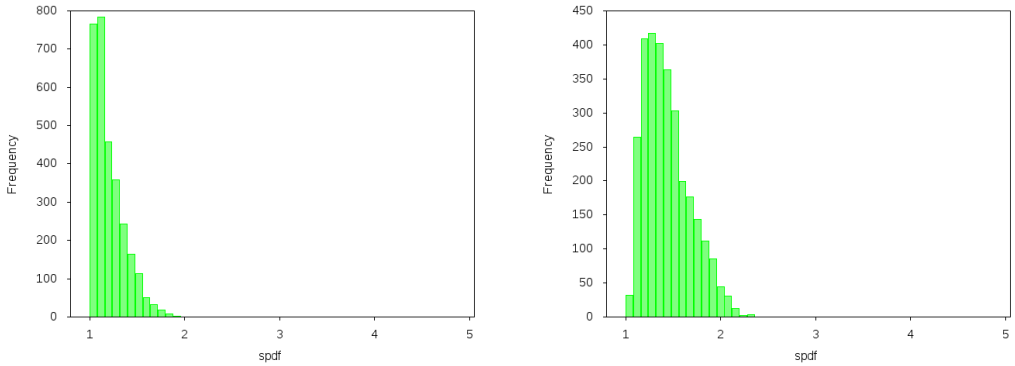


(g) Tasks not similar and $|\tau|$=100.



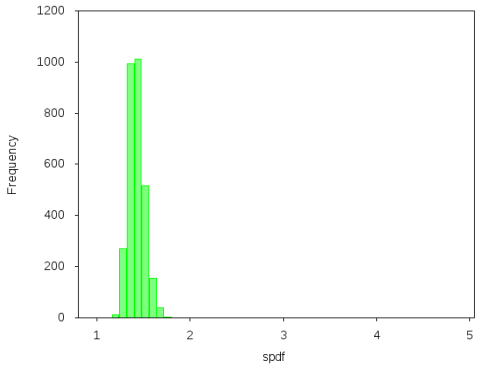
(h) Tasks not similar and $|\tau|$=1000.

Figure 1: Histogram for spdf for different ways of generating tasksets randomly for the case that T=∞.

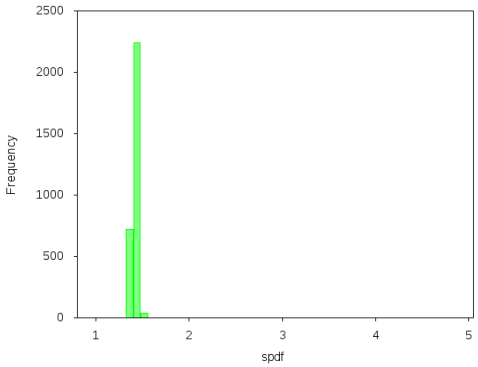


(a) Tasks similar and $|\tau|$=2.

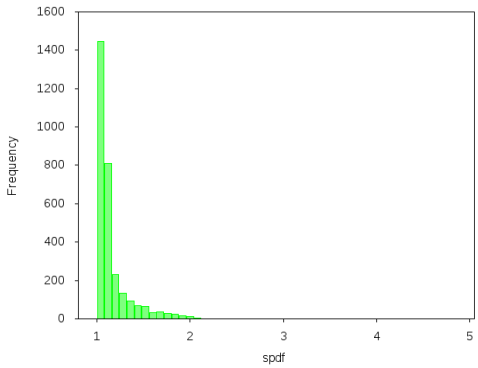
(b) Tasks similar and $|\tau|$=10.



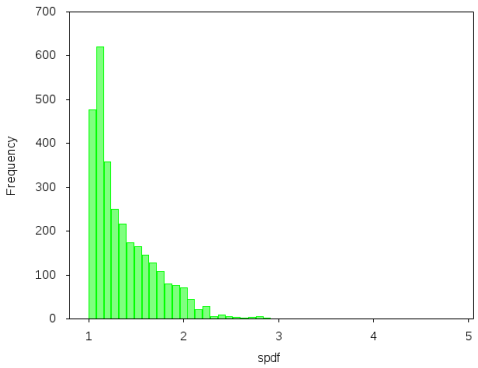
(c) Tasks similar and $|\tau|$=100.



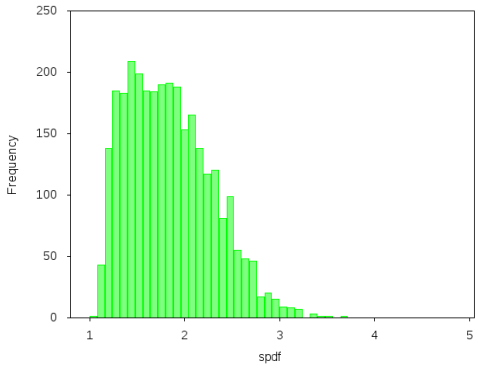
(d) Tasks similar and $|\tau|$=1000.



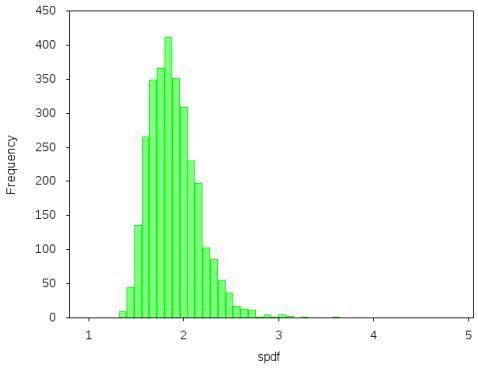
(e) Tasks not similar and $|\tau|$=2.



(f) Tasks not similar and $|\tau|$=10.

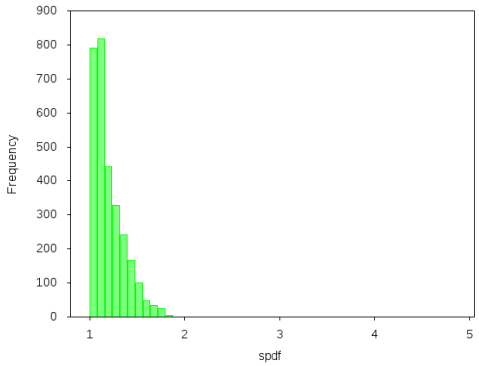


(g) Tasks not similar and $|\tau|$=100.

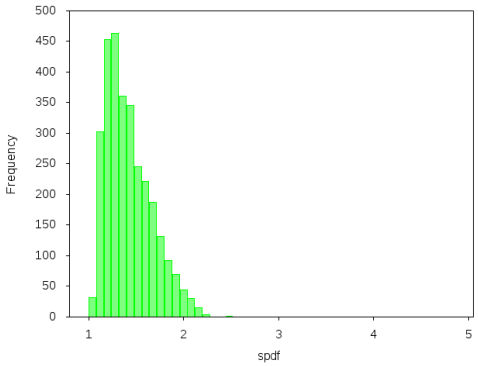


(h) Tasks not similar and $|\tau|$=1000.

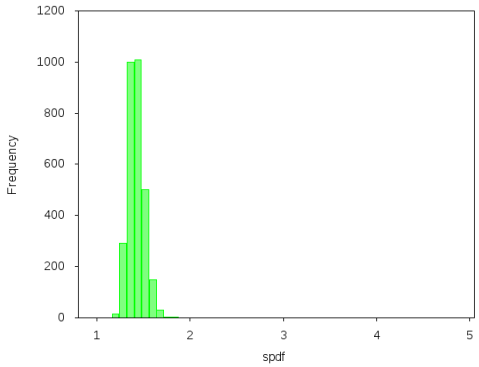
Figure 2: Histogram for spdf for different ways of generating tasksets randomly for the case that T is finite. Arbitrary-deadline sporadic tasks.



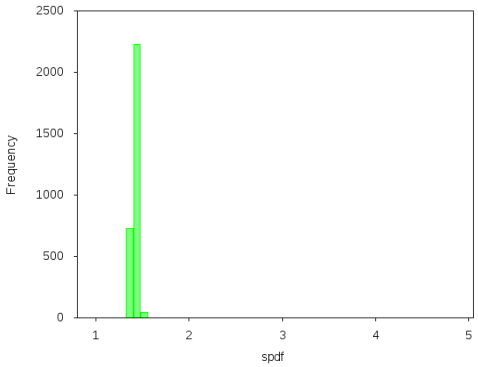
(a) Tasks similar and $|\tau|$=2.



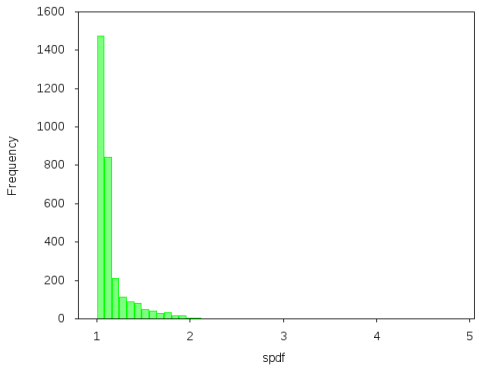
(b) Tasks similar and $|\tau|$=10.



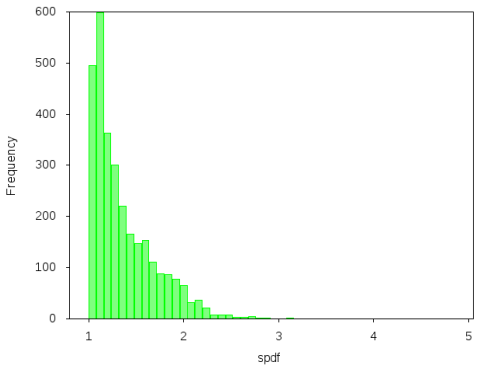
(c) Tasks similar and $|\tau|$=100.



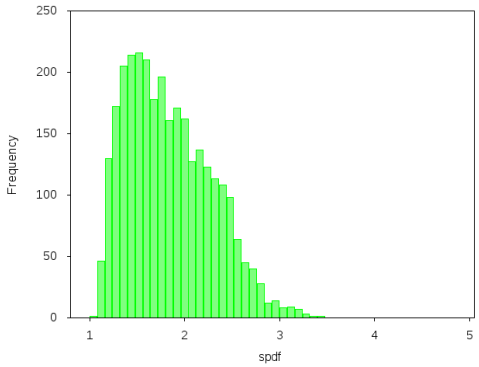
(d) Tasks similar and $|\tau|$=1000.



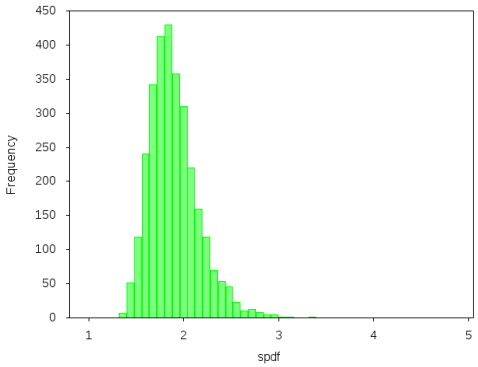
(e) Tasks not similar and $|\tau|$=2.



(f) Tasks not similar and $|\tau|$=10.



(g) Tasks not similar and $|\tau|$=100.



(h) Tasks not similar and $|\tau|$=1000.

Figure 3: Histogram for spdf for different ways of generating tasksets randomly for the case that T is finite. Constrained-deadline sporadic tasks.

6. EVALUATION FOR THE GENERAL CASE, IMPLICIT-DEADLINE SPORADIC TASKS

In this section, we consider the special case where for each task τ_i in τ, it holds that $C_i = T_i$.

Recall from Eq. 4 that

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{\min(D_i, T_i)}}{\max_{t>0}(\sum_{\tau_i \in \tau} \max(\lfloor \frac{t-D_i}{T_i} \rfloor + 1, 0) \times \frac{C_i}{t})} \quad (9)$$

Using $C_i = T_i$ on this expression yields that:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{T_i}}{\max_{t>0}(\sum_{\tau_i \in \tau} \max(\lfloor \frac{t-T_i}{T_i} \rfloor + 1, 0) \times \frac{C_i}{t})} \quad (10)$$

Rewriting yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{T_i}}{\max_{t>0}(\sum_{\tau_i \in \tau} \lfloor \frac{t}{T_i} \rfloor \times \frac{C_i}{t})} \quad (11)$$

It can be seen that the denominator is maximized for $t \to \infty$. This yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{T_i}}{\sum_{\tau_i \in \tau} \frac{C_i}{T_i}} \quad (12)$$

Simplifying yields:

$$\text{spdf}(\tau) = 1 \quad (13)$$

For this reason, there is no need to run experiments for implicit-deadline sporadic tasks.

7. COMMENTING ON THE EXPERIMENTAL RESULTS

Recall that spdf is a lower bound on how much faster a processor needs to be in order for a bandwidth-like scheme to meet deadlines. And recall that Figure 1, Figure 2, and Figure 3 provide histograms for spdf. From these figures, it can be seen that (i) the experimental results depend on the experimental setup, (ii) this lower bound on the penalty was never larger than 4.0, (iii) for one experimental setup, for each taskset, it was greater than 2.4, (iv) the histogram of this penalty appears to be unimodal, and (v) for implicit-deadline sporadic tasks, this lower bound on the penalty was exactly 1.

Let us now discuss two subfigures in Figure 1 in order to understand the results.

Consider Figure 1a. It shows an experimental setup with just two tasks so it is easy to see details. Consider the taskset with the largest observed spdf. By inspecting this taskset, we see that its spdf is 1.862861. We will now discuss how this result was obtained. The taskset contains 2 tasks and it turns out the denominator (of Eq. 7) is maximized for $j = 1$. For the taskset, it holds that $D_1 = 1.082638$. The taskset is as follows: $C_1 = 0.58$, $D_1 = 1.08$, $C_2 = 4.58$, $D_2 = 9.91$. Consider Eq. 7 again. Recall that there is a denominator with a max-expression. It turns out that for this taskset, this expression is maximized for $j = 1$. We also know that $D_1 = 1.082638$. Applying this knowledge on Eq. 7 yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{D_i}}{(\sum_{\tau_i \in \tau \ s.t. \ D_i \leq 1.082638} C_i)/1.082638} \quad (14)$$

We can compute the sum in the denominator; it is the sum of C for tasks with $D \leq 1.082638$. Using values yields that it is 0.58 and hence the denominator is 0.58/1.082638. This yields that the denominator is 0.535 and we have already seen that the numerator is 0.999. Hence,

$$\text{spdf}(\tau) = \frac{0.999}{0.535} \quad (15)$$

This yields: spdf = 1.86.

Consider Figure 1h. It shows the experimental setup where the largest observed spdf occurred. By inspecting this taskset, we see that its spdf is 3.899300. We will now discuss how this result was obtained. The taskset contains 1000 tasks and it turns out the denominator (of Eq. 7) is maximized for $j = 148$. For the taskset, it holds that $D_{148} = 21.94$. In order to give an idea of what the taskset looks like while not consuming too much space, we list below only the first task and also list tasks with $D \leq 21.94$ and with $C > 0.20$. The listing is as follows: $C_1 = 0.24$, $D_1 = 2.06$, ..., $C_{13} = 0.43$, $D_{13} = 4.12$, ..., $C_{30} = 0.59$, $D_{30} = 6.28$, ..., $C_{58} = 0.24$, $D_{58} = 8.70$, ... $C_{88} = 0.25$, $D_{88} = 12.25$, ... $C_{102} = 0.45$, $D_{102} = 14.28$, ... $C_{106} = 0.23$, $D_{106} = 15.03$, ... $C_{119} = 0.27$, $D_{119} = 16.71$, ... $C_{134} = 0.22$, $D_{134} = 19.56$, ... $C_{145} = 0.58$, $D_{145} = 21.05$, ... $C_{147} = 0.20$, $D_{147} = 21.81$. Consider Eq. 7 again. Recall that there is a denominator with a max-expression. It turns out that for this taskset, this expression is maximized for $j = 148$. We also know that $D_{148} = 21.94$. Applying this knowledge on Eq. 7 yields:

$$\text{spdf}(\tau) = \frac{\sum_{\tau_i \in \tau} \frac{C_i}{D_i}}{(\sum_{\tau_i \in \tau \ s.t. \ D_i \leq 21.94} C_i)/21.94} \quad (16)$$

We can compute the sum in the denominator; it is the sum of C for tasks with $D \leq 21.94$. Using values yields that it is 5.621 and hence the denominator is 5.621/21.94. This yields that the denominator is 0.2562 and we have already seen that the numerator is 0.999. Hence,

$$\text{spdf}(\tau) = \frac{0.999}{0.2562} \quad (17)$$

This yields: spdf = 3.899300.

8. RELATED WORK

The literature on hierarchical scheduling, composability, and compositionality is vast; here we only survey some of the previous work. When rate-monotonic scheduling was developed as a comprehensive framework, it was recognized that many systems have software whose resource consumption is hard to characterize; e.g., it is hard to find its worst-case execution time or minimum inter-arrival time. For this reason, reservation-based frameworks were developed (see for example [13]). The run-time behavior of such a framework is as follows: A task may be associated with a server task and this server task is scheduled as a normal task (for example with an execution time, often called budget, and a period)

and if a task is associated with a server task then it is only allowed to execute when the server task executes. In this way, if a task τ_i is in a server task and if τ_i experiences an execution overrun or arrives more often than expected then its impact on other tasks is bounded and it is bounded by the parameters of the server task. Later works created such reservation for EDF; one example of that is the constant-bandwidth server (CBS) [1].

Researchers realized that reservation-based frameworks can be used to form hierarchical scheduling; some work that did so with EDF include [14, 9]. With the focus on hierarchical scheduling, Feng and Mok developed [11] a resource model which states that a root scheduler supplies, in a time interval of duration t, at least $(t-\Delta^k)*\alpha^k$ units of execution to component k. Shin and Lee developed [15] another model where a component k is characterized by its period and execution time and with this, presented a so-called supply-bound function; this work could be used for components that use fixed-priority as a local scheduler or components that use EDF as a local scheduler. Later works have focused on resource sharing, specifically the question, if a task τ_{lock} executes within a critical section and its current budget has reached zero, what should the scheduler do? There may be other tasks that will request this critical section and these requests can be granted only if the task that currently holds that critical section (τ_{lock}) has released it. In order for the task (τ_{lock}) to release the critical section, it must be able to execute and in order for this task (τ_{lock}) to execute, it must have a current budget greater than zero. Different solutions for this have been developed; see for example [8, 7, 6]. Clearly, server parameters must be selected in order to ensure schedulability; this has been the focus of [2].

Recall that early in this paper, we pointed out that bandwidth-like schemes can suffer from poor performance and we have suggested that using the demand-bound function is an interface that does not suffer from this drawback. A similar point has been made in [12, 10]; they also discuss approximate policing of such an interface.

9. CONCLUSIONS

Bandwidth-like schemes can cause a performance penality. It was known that for certain tasksets, this performance penalty is infinite but it was not known how large this penalty is for average tasksets. Therefore, this paper generated tasksets randomly and computed spdf which is a lower bound on how much faster a processor needs to be in order to meet deadlines if a bandwidth-like scheme is used. Through the experimentation in this paper, it can be seen that (i) the experimental results depend on the experimental setup, (ii) this lower bound on the penalty was never larger than 4.0, (iii) for one experimental setup, for each taskset, it was greater than 2.4, (iv) the histogram of this penalty appears to be unimodal, and (v) for implicit-deadline sporadic tasks, this lower bound on the penalty was exactly 1.

Acknowledgment

Copyright 2016 ACM This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003264

10. REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, 1988.

[2] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT*, 2004.

[3] B. Andersson. A pseudo-medium-wide 8-competitive interface for two-level compositional real-time scheduling of constrained-deadline sporadic tasks on a uniprocessor. In *CRTS*, 2009.

[4] B. Andersson. A preliminary idea for an 8-competitive, log2 DMAX + log2 log2 (1/U) asymptotic-space, interface generation algorithm for two-level hierarchical scheduling of constrained-deadline sporadic tasks on a uniprocessor. In *CRTS*, 2010.

[5] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. In *Real-Time Systems*, pages 301–324, 1990.

[6] M. Behnam, T. Nolte, M. Asberg, and R. J. Bril. Overrun and skipping in hierarchically scheduled real-time systems. In *RTCSA*, 2009.

[7] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In *EMSOFT*, 2007.

[8] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS*, 2006.

[9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, 1997.

[10] F. Dewan and N. Fisher. Efficient admission control for enforcing arbitrary real-time demand-curve interfaces. In *RTSS*, 2012.

[11] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *RTSS*, 2002.

[12] P. Kumar, J.-J. Chen, and L. Thiele. Demand bound server: Generalized resource reservation for hard real-time systems. In *EMSOFT*, 2011.

[13] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *RTSS*, 1987.

[14] G. Lipari and S. K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *RTAS*, 2001.

[15] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, 2003.