Efficient Admission Control for Enforcing Arbitrary Real-Time Demand-Curve Interfaces

Farhana Dewan Nathan Fisher
Department of Computer Science
Wayne State University
{farhanad, fishern}@cs.wayne.edu

***Abstract*—Server-based resource reservation protocols (e.g., periodic and bandwidth-sharing servers) have the advantage of providing temporal isolation between subsystems co-executing upon a shared processing platform. For many of these protocols, temporal isolation is often obtained at the price of over-provisioned reservations. Other more fine-grained approaches such as real-time calculus permit a precise characterization of the resources required by a subsystem via demand-curve interfaces. However, an important, unsolved challenge for subsystems specified by such interfaces is the development of efficient enforcement techniques to guarantee temporal isolation between the subsystems. Admission control algorithms can be used in this regard to ensure that the cumulative subsystem demand never violates the demand-curve specified by the interface. In this paper, we address the challenge by designing admission controllers for complex, arbitrary demand-curve interfaces and proposing enforcement techniques. First we propose an exact algorithm and show that its complexity is infeasible for long-running systems. To address this drawback, we design an approximate algorithm and associated enforcement techniques to handle unpredictable execution times. We validate, via simulations, that our approximate approach is significantly more efficient than the exact approach with only minor decrease in the accuracy of the admission controller.**

***Index Terms*—demand-curve interfaces; admission control; approximation algorithm; temporal isolation.**

I. Introduction

Recent real-time and embedded systems research is increasingly trending towards open environments [1] due to the ease of portability and integration of independently-developed subsystems upon a shared platform. Such systems are mostly implemented via *resource partitions* [2] which ensure that each subsystem is guaranteed access to a shared computational resource without interference from other subsystems in the system. Examples of such real-time open environments include automotive subsystems, each with strict temporal requirements, integrated together on a shared electronic control unit [3]. For such systems with specific interface requirements, one fundamentally important challenge is to ensure *temporal isolation* between subsystems. That is, if any of the subsystems malfunctions or violates the interface, its behavior should be policed such that no other sub-system is affected. However, such temporal isolation should not be achieved at the expense of inefficient resource usage (i.e., over-provisioning of resources should be avoided, if possible).

A popular approach for achieving temporal isolation between subsystems in a real-time open environment is to use server-based interfaces; each subsystem may execute within a system-provided server that allocates the resource based upon the subsystem's real-time interface. The main disadvantage of these approaches is that they require some over-provisioning of resources to guarantee temporal isolation [4]. A more precise alternative to server-based approaches is to use a *demand-curve interface* to specify the amount of computational resources (over any interval of time) that the subsystem will require. The well known real-time calculus (RTC) frameworks [5], [6] are examples of this approach. However, strict temporal isolation is currently difficult to achieve between subsystems specified by demand-curve interfaces, as there is no known online "policing" protocol for ensuring that a system does not violate an arbitrarily-specified demand-curve interface [7].

In this paper, we address this lack of interface-policing protocols for the more precise demand-curve interface models. To achieve this, we propose admission controllers for subsystems comprising of aperiodic hard-real-time jobs. These admission controllers will ensure temporal isolation by checking whether a newly-arrived job may be admitted for execution in a subsystem without violating its demand-curve interface. Furthermore, we describe the implementation of an associated server which can enforce (at runtime) the temporal isolation once a job is admitted to the system. Our primary design goal is the development of admission controllers that are both theoretically and practically efficient; i.e., we can prove tight, polynomial bounds on computation complexity and observe low overhead in an implementation.

Our contributions can be listed as follows:

- We propose an exact admission control algorithm for *monotonic ascending deadline (*MAD*)* [8] aperiodic jobs, and prove its correctness (Section IV). We also argue that the exact approach is not computationally feasible for long-running systems.
- To address the infeasibility of the exact approach, we devise an efficient approximate algorithm for the admission control of MAD jobs (Section V).
- We provide mechanisms for enforcing temporal isolation between admitted jobs by designing a lightweight server within which each job may be executed (Section VI-B).

- We implement each of our proposed admission controllers and show that our approximate admission controller is both efficient and precise (in comparison to the exact) via simulation (Section VII).
- We give a straightforward extension of our approximate admission controller for MAD jobs to arbitrary aperiodic jobs (Section VIII).

Before describing the details of our contributions, we first present the real-time job and interface models assumed in this paper (Section II) and discuss related work (Section III).

II. Model and Notations

In this section, we introduce the job and interface models that characterize a subsystem and provide a formal definition of the problem addressed in this paper.

A. Job Model

We assume that jobs can arrive aperiodically for a subsystem. Each aperiodic job j_i is characterized by an *arrival time* A_i, a *worst case execution requirement* E_i, and a *relative deadline* D_i; a job j_i is denoted by the three-tuple (A_i, E_i, D_i). We also denote the *absolute deadline* for j_i as $d_i \overset{\text{def}}{=} A_i + D_i$. A job set $J = \{j_1, j_2, \ldots\}$ is a finite set of jobs indexed in order of increasing arrival time (i.e., for $1 \le i < |J| : A_i \le A_{i+1}$).

We assume that job parameters are revealed to a subsystem only upon job arrival; i.e., a subsystem does not have knowledge of future job arrivals. We call a job j_i *active* at time t, if $t \in [A_i, A_i + D_i)$. Let N be the maximum number of active jobs in the subsystem at any given time.

In general, we place no restriction on the parameters of jobs (except being non-negative numbers) that arrive in the system. However, as a starting point in our development of an efficient admission controller, we restrict ourselves to *monotonic absolute deadline* (MAD) [8] job arrivals first. For MAD jobs, if job j_i arrives before job j_k, then j_i's absolute deadline must occur before j_k's absolute deadline; more formally, $A_i \le A_k \Leftrightarrow d_i \le d_k$. This assumption is appropriate for subsystems that have either a single job-generation stream or jobs with identical relative deadlines. For a more general setting, we refer to jobs that arrive in any possible (non-MAD) sequence as an *arbitrary job arrivals* and give a straightforward extension of our MAD admission controller to arbitrary job arrivals in Section VIII.

Given a set of jobs J, we now describe how to accurately quantify the maximum workload over any interval.

Definition 1 (Demand): For any J and $t_1, t_2 \in \mathbb{R} : 0 \le t_1 < t_2$, the function $\mathsf{demand}(J, t_1, t_2)$ represents the maximum cumulative execution requirement of all jobs in J that have both an arrival time and deadline in the interval $[t_1, t_2]$.

$$\mathsf{demand}(J, t_1, t_2) = \sum_{\substack{j_i \in J: \\ (A_i \ge t_1)\ \wedge\ (d_i \le t_2)}} E_i. \tag{1}$$

B. Interface Model

In compositional real-time systems, an interface exposes to the system the temporal requirements of a subsystem. We denote the interface of the subsystem for which we are designing an admission controller as Λ. Numerous real-time interface models have been proposed in recent years (e.g., the periodic resource model [9], real-time calculus [6], etc.); however, in this paper, we consider Λ to be from a non-specific interface model. Our only requirement is that interface model permits a characterization of the admissible demand over intervals of time. Throughout this paper, we assume that the system designer has already generated and specified the interface Λ. The challenge of generating and composing demand-curve interfaces is important, but orthogonal to the problem we address in this paper. (See Thiele et al. [6] for a discussion of these issues.)

The following definition gives a general specification of the maximum allowable demand of a subsystem over time.

Definition 2 (Arbitrary Demand-Bound Curve): An arbitrary demand-bound curve of interface Λ gives an upper bound on the total demand of the set of jobs J admitted by a subsystem. We denote the demand-bound curve for any interval of positive length t as $\mathsf{dbi}(\Lambda, t)$. Formally, the dbi ensures that the following condition holds

$$\forall t_1, t_2 \in \mathbb{R} : (0 \le t_1 < t_2) :: \mathsf{demand}(J, t_1, t_2) \le \mathsf{dbi}(\Lambda, t_2 - t_1). \tag{2}$$

The dbi is a right continuous, piecewise linear, non-negative, and non-decreasing function of interval lengths $t \in \mathbb{R}_{\ge 0}$.

The above definition is applicable to existing real-time interface models such as real-time calculus (RTC). The supply-bound function of periodic resource model can be considered as a dbi. While the existing results for subsystems scheduled by earliest-deadline-first or fixed-priority upon server-based interfaces [9], [10] can be used as admission controllers, we do not assume any specific scheduling algorithm and focus only on ensuring that the underlying subsystem does not violate the specified demand interface.

C. Problem Statement

When a job is admitted into a subsystem with real-time interface Λ, we must ensure that the total demand over any interval does not exceed the demand-curve specified by $\mathsf{dbi}(\Lambda, \cdot)$. Formally, the objective of our admission controller is as follows.

> Given an interface Λ and a set of jobs J that have previously been admitted to by time t (i.e., Equation 2 holds for J). Let j_k be a job that arrives to the subsystem at time t. The objective of the admission controller is to determine whether Equation 2 continues to hold for the job set $J \cup \{j_k\}$. If the above condition holds, then job j_k may be admitted into the subsystem.

III. Related Work

Several interface-based frameworks have been proposed [11]–[13] for subsystems of a compositional real-time system. For these frameworks the compositional design is based on real-time interfaces [14] and the analysis is based on real-time calculus (RTC) [5], [6]. Wandeler and Thiele [11]–[13] proposed a model where interfaces for the subsystems are “assumed”, and the system “guarantees” the interface to the subsystem. Ensuring temporal isolation among subsystems is not trivial in this model.

Linear-time exact admission tests for scheduling periodic and aperiodic jobs have been proposed in [15], [16]. Lipari and Buttazzo [17] proposed *Bandwidth Sharing Server (BSS)* algorithm which provides precise isolation between subsystems. In [18], this model has been extended to support aperiodic servers with different subsystem level schedulers. Using a similar approach, Andersson and Ekelin [19] proposed an $O(\log N)$ exact admission controller for aperiodic and periodic jobs in a non-compositional setting. Dewan and Fisher [20] extended their techniques to apply to admission control for simple (single-step) demand-curve interfaces.

A recent paper by Kumar et al. [21] has proposed a Demand-Bound Server (DBS) for scheduling jobs according to a demand-curve interface. The proposed server successfully achieves the goal of providing temporal isolation between subsystems specified precisely by a demand-curve. However, the approach has fundamental differences with our proposed approach. First, Kumar et al. do not provide an admission controller for DBS; thus, if a subsystem incorrectly generates workload which exceeds the specified demand curve, the over-allocation error would only become apparent when the subsystem misses a deadline. In our approach, we seek to identify jobs that exceed a subsystem’s demand-curve interface before they are admitted to the scheduler. This approach permits a subsystem designer to identify and recover from potential over-allocation errors early before a temporal violation has occurred. The second fundamental difference is that Kumar et al. assume that jobs are scheduled in FCFS order. Our general approach makes no assumptions regarding the underlying execution of admitted jobs; we only guarantee that the set of admitted jobs does not violate the demand-curve interface. In Kumar et al. [21], while a general mathematical model is presented for arbitrary demand curves, the server algorithm has only been specified for a very simple periodic demand function corresponding to the demand of a single periodic task. In this paper, we perform admission control and ensure temporal isolation for an arbitrarily-complex demand curve.

IV. Exact Admission Control for MAD Jobs

In this section, we will give a straightforward algorithm for exact admission control for an arbitrary demand-curve interface. We will first give the algorithm and prove its correctness. Next, we will illustrate that the exact approach is not computationally tractable for long-running online systems. Thus, we require more sophisticated techniques for efficient admission control, as will be explored in the next section.

A. Algorithm Description

```
ExactAC-Init()
▷ $S$ (initially empty) will keep a set of $(x, y)$ values
$S \leftarrow \emptyset$; $d_{\text{last}} \leftarrow 0$

ExactAC($\Lambda, j_k$)
$\delta_x \leftarrow d_k - d_{\text{last}}$; $\delta_y \leftarrow E_k$
▷ Insert a point at the beginning of $S$
$S \leftarrow \{(D_k - \delta_x, 0)\} \cup S$
for $(x, y) \in S$
        if dbi$(\Lambda, x + \delta_x) < (y + \delta_y)$
                ▷ Delete previously inserted point.
                $S \leftarrow S \setminus \{(D_k - \delta_x, 0)\}$
                Reject $j_k$.
end for
Accept $j_k$.
▷ Shift points in $x, y$-plane.
for $(x, y) \in S$
        $S \leftarrow S \setminus \{(x, y)\}$
        $S \leftarrow S \cup \{(x + \delta_x, y + \delta_y)\}$
end for
$d_{\text{last}} \leftarrow d_k$
```

Fig. 1. Pseudo-code for exact admission control of MAD jobs with arbitrary demand-curve Λ.

Let us consider the scenario where we have already admitted a set of MAD jobs J and are attempting to determine whether we can admit a new job j_k (where j_k has later arrival time and deadline than all previously-admitted jobs). Observe that an exact admission control algorithm is conceptually relatively straightforward: to check whether a job j_k can be admitted, calculate the change in demand over every interval $[t_1, t_2]$ and check the inequality of Equation 2. However, a naive implementation of this idea would require the evaluation of Equation 2 for an infinite number of intervals. A practical (finite-time) implementation of the exact algorithm can be developed from the observation that only a finite number of intervals must be checked for determining whether to admit j_k: intervals that begin at the arrival of some job of J and end at d_k. The exact

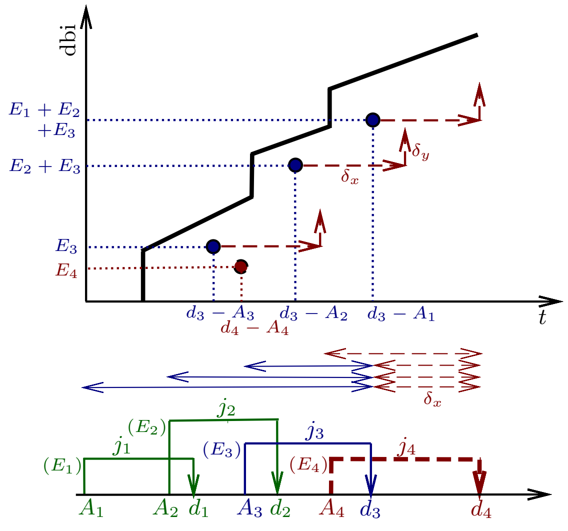


Fig. 2. Illustration of exact admission control algorithm.

algorithm shown in ExactAC and ExactAC-Init (Figure 1) maintains an ordered set S of intervals each specified by a demand pair (x, y) where x corresponds to the length the interval and y corresponds to demand over that interval. The set S is ordered in non-decreasing value of the x-coordinate for each pair. The variable d_{last} stores last accepted job's absolute deadline. The demand pair can be mapped to a point in the cartesian plane (Figure 2). Furthermore, we will show later that it is only necessary to store demand pairs in S where the interval length x corresponds to the difference between the arrival time of some accepted job of J and d_{last} as shown in Figure 2.

When a new job j_k arrives to the system, the demand of the intervals $(d_k - A_i), \forall j_i \in J$ needs to be checked. In Line 2 of ExactAC, an (x, y) interval corresponding to j_k is added to the set. Since jobs arrive in MAD order, the intervals ending at d_k can be obtained by incrementing x values in the set S by $\delta_x = d_k - d_{\text{last}}$, and the demand over these intervals can be obtained by incrementing y values in the set by $\delta_y = E_k$ amount. For all the new intervals with the increment, the algorithm checks if the demand in the interval is less than the dbi (Line 3 to Line 7). If for any interval this condition is violated, j_k is rejected and the point corresponding to it is removed from S. When the condition holds for all the points in S, j_k is admitted and all the points in S are updated by the incrementing (δ_x, δ_y) amount (Line 9 to Line 12). Figure 2 illustrates the algorithm after j_4 arrives to the system. The set S consists of the intervals ending at d_3 (blue points shown in the plot) prior to the arrival of j_4. When j_4 arrives, after successful admission test, the demand pair $(d_4 - A_4, E_4)$ is added (red point in the figure) in S, and all the existing intervals are shifted by $(\delta_x = d_4 - d_3, \delta_y = E_4)$ amount.

Exact admission control for arbitrary demand-curve is computationally linear to the number of jobs that have arrived in the system. Since all past intervals need to be verified, the number of such intervals become intractable with time. In previous work, we have addressed this problem by considering a single-step demand interface [20] (i.e., dbi is linear). However, this previous approach is not applicable to arbitrary demand curves. The main challenge for arbitrary demand interface is the difference in demand and the interface for any interval changes non-linearly over time. More specifically, for a same increase in interval length and same increase in demand, the minimum demand difference for an interval will not necessarily represent minimum demand difference after increment.

B. Proof of Correctness

We now provide the main theorem which state that ExactAC is correct and exact with respect to the problem definition of Section II-C.

Theorem 1: Given a set of previously-admitted jobs J, the procedure ExactAC(Λ, j_k) returns "Accept", if and only if, j_k may be admitted without $J \cup \{j_k\}$ violating Λ.

We need to prove the next three lemmas in order to prove the theorem. Due to space constraints, we only state the lemmas which have similar proofs in this section and the next section. A detailed proof can be found in the appendix of the extended version of this paper [22]. The first lemma shows that we only need to check a finite number (with respect to the number of jobs in J) to determine whether Equation 2 is satisfied.

Lemma 1: For any set of MAD jobs J, Equation 2 is true, if and only if,

$$\forall j_i, j_\ell \in J : d_i \leq d_\ell :: \mathsf{demand}(J, A_i, d_\ell) \leq \mathsf{dbi}(\Lambda, d_\ell - A_i). \tag{3}$$

Proof: The "only if" direction of the lemma is trivial as A_i and d_ℓ are elements of $\mathbb{R}$ and $A_i \leq d_\ell$. Thus, Equation 2 directly implies Equation 3.

For the "if" direction of the lemma, we will assume that Equation 3 is true, but Equation 2 is false for some $t_1, t_2 \in \mathbb{R}$ where $0 \leq t_1 < t_2$; that is,

$$\mathsf{demand}(J, t_1, t_2) > \mathsf{dbi}(\Lambda, t_2 - t_1). \tag{4}$$

If $J = \emptyset$, then the demand over any interval is zero; since dbi is non-negative for all positive inputs, this leads to a contradiction of Equation 4. So, it must be that $J \neq \emptyset$. Let A_0 and d_0 denote zero and $A_{|J|+1}$ and $d_{|J|+1}$ denote ∞. Consider two partitions of the interval $[0, \infty)$ into two sets of subintervals $(A_{i-1}, A_i]$ where $1 \leq i \leq |J| + 1$ and $[d_\ell, d_{\ell+1})$ where $0 \leq \ell \leq |J|$. Since $0 < t_1$, there exists some $i : (1 \leq i \leq |J| + 1)$ where $t_1 \in (A_{i-1}, A_i]$. Observe that $\mathsf{demand}(J, t_1, t_2) = \mathsf{demand}(J, A_i, t_2)$, since no jobs arrive in the interval (A_{i-1}, A_i). Similarly, there exists some $\ell : (0 \leq \ell \leq |J|)$ where $t_2 \in [d_\ell, d_{\ell+1})$ and $\mathsf{demand}(J, A_i, t_2) = \mathsf{demand}(J, A_i, d_\ell)$. Thus, $\mathsf{demand}(J, t_1, t_2) = \mathsf{demand}(J, A_i, d_\ell)$. By Equation 3, $\mathsf{demand}(J, t_1, t_2) \leq \mathsf{dbi}(\Lambda, d_\ell - A_i)$ is true. Since $d_\ell \leq t_2$, $t_1 \leq A_i$, and dbi is monotonically non-decreasing, it must be that $\mathsf{dbi}(\Lambda, d_\ell - A_i) \leq \mathsf{dbi}(\Lambda, t_2 - t_1)$. These last two statements together imply that $\mathsf{demand}(J, t_1, t_2) \leq \mathsf{dbi}(\Lambda, t_2 - t_1)$ which contradicts Equation 4. Thus, the lemma is true. ■

The next lemma shows the correspondence between points stored in S and the intervals ending at the deadline of the last admitted job (i.e., d_{last}).

Lemma 2: After the call to ExactAC-Init() and the k'th invocation of ExactAC(Λ, j_k) where $k \in \mathbb{N}$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the algorithm. For each job $j_\ell \in J_k$, there exists $(x, y) \in S$ such that x equals $d_{\text{last}} - A_\ell$ and y equals $\mathsf{demand}(J_k, A_\ell, d_{\text{last}})$. Furthermore, d_{last} equals $\max\{\{0\} \cup \{d_\ell \mid j_\ell \in J_k\}\}$.

Proof: We prove the lemma by induction on k.

Base Case: When $k = 0$, ExactAC-Init() has been invoked and thus no jobs have been admitted; i.e, $J_0 = \emptyset$. The lemma is clearly true as S is initialized to $\emptyset$ and d_{last} is initialized to zero.

Inductive Hypothesis: Assume that the lemma holds for each i ($i = 1, 2, \ldots, k-1$) successive calls to $\textsc{ExactAC}(\Lambda, \cdot)$.

Inductive Step: We must show that the lemma holds for the k'th call to $\textsc{ExactAC}(\Lambda, j_k)$. The admission controller can either return "accept" or "reject". Let us first consider the case that $\textsc{ExactAC}(\Lambda, j_k)$ returns "reject". Then, J_{k-1} is identical to J_k and d_{last} is not changed by any instruction in the execution path to "reject". Thus, by the inductive hypothesis, the lemma obviously continues to hold as the state is identical to after the call to $\textsc{ExactAC}(\Lambda, j_{k-1})$.

Now, consider the case when $\textsc{ExactAC}(\Lambda, j_k)$ returns "accept". Line 13 of the procedure sets d_{last} equal to d_k; Let the updated value of d_{last} and S be denoted by $d_{\text{last}}^{\text{new}}$ and S^{new} respectively. Let $d_{\text{last}}^{\text{old}}$ and S^{old} denote the value of the d_{last} and S variables, prior to $\textsc{ExactAC}(\Lambda, j_k)$. By the inductive hypothesis, for each job $j_\ell \in J_{k-1}$, there exists $(x, y) \in S^{\text{old}}$ such that x equals $d_{\text{last}}^{\text{old}} - A_\ell$ and y equals $\mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}}^{\text{old}})$. The *for-loop* of Lines 9 to 12 shifts each point (x, y) to the right by $\delta_x = d_{\text{last}}^{\text{new}} - d_{\text{last}}^{\text{old}}$ and up by $\delta_y = E_k$. Thus, each $(x, y) \in S^{\text{old}}$ that corresponds to $j_\ell \in J_{k-1}$ is now $(x+\delta_x, y+\delta_y) \in S^{\text{new}}$. Furthermore, $x+\delta_x$ equals $(d_{\text{last}}^{\text{new}} - d_{\text{last}}^{\text{old}}) + d_{\text{last}}^{\text{old}} - A_\ell = d_{\text{last}}^{\text{new}} - A_\ell$ and $y + \delta_y$ equals $\mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}}^{\text{old}}) + E_\ell$. The last expression is equivalent to $\mathsf{demand}(J_k, A_\ell, d_{\text{last}}^{\text{new}})$ since increasing the interval length by δ_x includes only the new job j_k in the interval $[A_\ell, d_{\text{last}}^{\text{new}}]$. Finally, adding the point $\{(D_k - \delta_x, E_k - \delta_y)\}$ in Line 2 and shifting by δ_x and δ_y is equivalent to adding (D_k, E_k) which equals $(d_{\text{last}}^{\text{new}} - A_k, \mathsf{demand}(J_k, A_k, d_{\text{last}}^{\text{new}}))$. Thus, the lemma holds for J_k. ■

Finally, to determine whether to admit j_k, we only need to check intervals ending at d_k (and that begin at some arrival time). Since we have stored all intervals that end at d_{last} ($\leq d_k$) in S, we may update the points in S by shifting them to the right $\delta_x = d_k - d_{\text{last}}$ and upwards by $\delta_y = E_k$. We also add a point (D_k, E_k) which corresponds to job j_k's demand over its arrival and deadline. A new or newly-shifted point will be above $\mathsf{dbi}(\Lambda, \cdot)$, if and only if, the corresponding interval ending at d_k violates Equation 2. This observation is formalized in the next lemma; thus, the algorithm returns "Accept", if and only if, it is safe to do so.

Lemma 3: After the call to $\textsc{ExactAC-Init}()$ and the k'th invocation of $\textsc{ExactAC}(\Lambda, j_k)$ where $k \in \mathbb{N}$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the algorithm. It must be that Equation 2 holds for J_k and Λ. Furthermore, if job j_k was rejected by the admission controller (i.e., $\textsc{ExactAC}(\Lambda, j_k)$ returns "reject" and $j_k \notin J_k$), then Equation 2 is false for $J_{k-1} \cup \{j_k\}$.

Proof: We prove the lemma by induction on k.

Base Case: When $k = 0$, $\textsc{ExactAC-Init}()$ has been invoked and thus no jobs have been admitted; i.e, $J_0 = \emptyset$. Clearly, $\mathsf{demand}(J_k, t_1, t_2)$ is zero for all valid choices of t_1 and t_2. Therefore, Equation 2 is trivially true.

Inductive Hypothesis: Assume that Equation 2 holds for each i ($i = 1, 2, \ldots, k-1$) successive calls to $\textsc{ExactAC}(\Lambda, \cdot)$. That is, J_i satisfies Equation 2. Furthermore, if job j_i is rejected, Equation 2 is false for $J_{i-1} \cup \{j_i\}$.

Inductive Step: We must show that, given the inductive hypothesis, 1) Equation 2 holds for J_k, and 2) if job j_k is rejected, then Equation 2 is false for $J_{k-1} \cup \{j_k\}$. During the call $\textsc{ExactAC}(\Lambda, j_k)$, the admission control can either return "accept" or "reject". Let us first consider the case that $\textsc{ExactAC}(\Lambda, j_k)$ returns "reject". Then, J_{k-1} is identical to J_k; since Equation 2 holds for J_{k-1}, by the inductive hypothesis, it obviously continues to hold for J_k. To see that Equation 2 is false for $J_{k-1} \cup \{j_k\}$, let S^{old} be the value of S before $\textsc{ExactAC}(\Lambda, j_k)$ and S^{new} be the value after the procedure call. δ_x is set to $d_{\text{last}}^{\text{old}} - d_k$ and δ_y is set to E_k. Observe that Line 4 must have evaluated to "true" for some $(x, y) \in S^{\text{old}} \cup \{(D_k - \delta_x, E_k - \delta_y)\}$. If (x, y) equals $(D_k - \delta_x, E_k - \delta_y)$, then $\mathsf{dbi}(\Lambda, D_k) < E_k$ which implies $\mathsf{dbi}(\Lambda, d_k - A_k) < \mathsf{demand}(J_{k-1} \cup \{j_k\}, A_k, d_k)$; in this case, Equation 2 is violated. Otherwise, if $(x, y) \in S^{\text{old}}$, then $\mathsf{dbi}(\Lambda, x + \delta_x) < y + \delta_y$ implies that $\mathsf{dbi}(\Lambda, d_k - A_\ell) < \mathsf{demand}(J_{k-1} \cup \{j_k\}, A_\ell, d_k)$ for some $j_\ell \in J_{k-1}$ by Lemma 2.

Now consider the case when $\textsc{ExactAC}(\Lambda, j_k)$ returns "accept". Lemma 2 implies that each $(x, y) \in S$ corresponds to $(d_k - A_\ell, \mathsf{demand}(J_k, A_\ell, d_k))$ for some job $j_\ell \in J_k$. The fact that the j_k was accepted implies Line 4 was satisfied for each (x, y) and thus $x \geq y$; that is,

$$\forall j_\ell \in J_k, \mathsf{dbi}(\Lambda, d_k - A_\ell) \geq \mathsf{demand}(J_k, A_\ell, d_k). \quad (5)$$

The inductive hypothesis implies that prior to the invoke of $\textsc{ExactAC}(\Lambda, j_k)$, Equation 2 held; therefore,

$$\begin{aligned} &\forall j_i, j_j \in J_{k-1} : d_i \leq d_j :: \\ &\mathsf{demand}(J_{k-1}, A_i, d_j) \leq \mathsf{dbi}(\Lambda, d_j - A_i). \end{aligned} \quad (6)$$

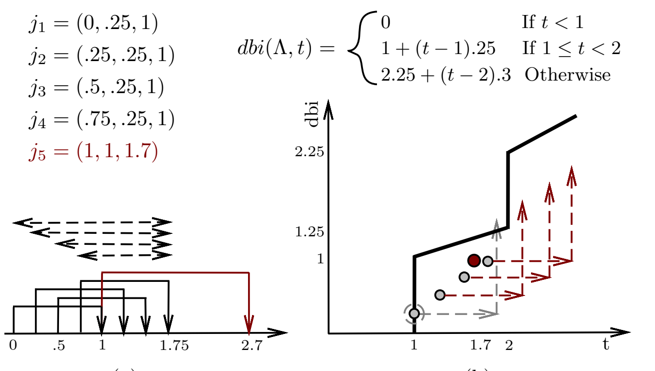
Since d_k is later than any other d_i in J_{k-1}, j_k does not increase the demand in the above term $\mathsf{demand}(J_{k-1}, A_i, d_j)$. Thus, the condition still holds when the left-hand side of the inequality is replaced with $\mathsf{demand}(J_k, A_i, d_j)$. This observation regarding Equation 6 along with Equation 5 satisfy the supposition of Lemma 1 for the job set J_k; therefore, Equation 2 is satisfied for J_k. ■

The combination of Lemmas 1, 2, and 3 immediately imply Theorem 1.

C. Inadequacy of Naive Reduction in Intervals

A natural idea for reducing the time complexity of the exact admission controller would be to discard some of the points of S. Unfortunately, we will see that this idea will lead to an incorrect admission controller. In this subsection, we give a job set J such that if any of the points in S (corresponding jobs in J) is discarded, there exist future job arrivals that will cause a violation of Equation 2, but the admission controller will not detect this violation.

Example 1: Consider a system that has admitted a set of jobs $J = \{j_1, \ldots j_4\}$ as shown in Figure 3(a). The demand-interface and the intervals corresponding to points in S (shown in grey) are illustrated in Figure 3(b). For any of the points in



$j_1 = (0, .25, 1)$
$j_2 = (.25, .25, 1)$
$j_3 = (.5, .25, 1)$
$j_4 = (.75, .25, 1)$
$j_5 = (1, 1, 1.7)$
$dbi(\Lambda, t) = \begin{cases} 0 & \text{If } t < 1 \\ 1 + (t-1).25 & \text{If } 1 \le t < 2 \\ 2.25 + (t-2).3 & \text{Otherwise} \end{cases}$
dbi
2.25
1.25
1
0
.5
1
1.75
2.7
1.7
2
t
(a)
(b)

Fig. 3. Naive reduction in intervals might cause violation of dbi.

S, if we discard it, we can construct a new job with parameters such that it would have caused the discarded point to exceed the dbi after shifting it towards right ($\delta_x = d_k - d_{last}$) and upwards ($\delta_y = E_k$). We will show that for the dbi and each of the intervals in Figure 3, a job can be constructed with parameters such that only one interval violates the dbi after the shift. In other words, if we discard one interval, the demand-interface violation might be undetected for the new job, which will lead to an incorrect admission controller.

First, consider that we have discarded the point corresponding to $[A_4, d_4] = [.75, 1.75]$ (the point within circle in Figure 3(b)). The point corresponding to this interval has the largest difference with dbi after the arrival of j_4. Let the next job, $j_5(A_5, E_5, D_5) \equiv (1, 1, 1.7)$, arrive into the system. The demand for the new interval $[1, 2.7]$ is 1, thus the point corresponding to this interval is below the dbi. Now all the other points $[A_i, d_5], i = 1 \ldots 3$ will be shifted by $(\delta_x, \delta_y) \equiv (2.7 - 1.75, 1)$, and they are safely below the dbi. But the point corresponding to $[A_4, d_4]$ interval would have moved beyond the dbi as dbi$(1 + 0.95) = 1.2375 \le 0.25 + 1$. Since this point is discarded, the system will admit j_5, which will eventually lead to a violation of demand curve interface. Similarly, if we discard $[A_3, d_4]$ interval, and insert a new job $j_5 = (1, 0.7, 1.5)$, the demand for the discarded interval will exceed dbi only, and other intervals will stay below dbi after the shift. Again, let us assume that $[A_2, d_4]$ interval is discarded, and a new job arrives with parameters $j_5 = (1, 0.5, 1.2)$, then the increment in all the other intervals will not violate the dbi, and only demand for $[A_2, d_4]$ interval will violate the dbi. Finally, consider we have discarded the $[A_1, d_4]$ interval and a new job $j_5 = (1, 0.25, 0.95)$ arrives in the system. Here also all the other intervals with their increment will satisfy the dbi and falsely accept j_5 to the system, whereas if we have stored the $[A_1, d_4]$ interval, the interface violation would have been detected. ■

This example illustrates that there exist sequences of job arrivals and dbi such that no point may naively be discarded. Therefore, simply throwing away intervals is not a solution to reducing the complexity of the admission controller.

V. Approximate Admission Control for MAD Jobs

As mentioned in the previous section, the exact admission controller is intractable for long-running systems. In this section we propose an approximate solution to efficiently perform admission control for MAD jobs. In our proposed approach, we reduce the number of intervals (points) using a more sophisticated approach than just naively dropping intervals as illustrated in the example of Section IV-C. We achieve our reduction in the time complexity for admission control via four main steps:

1) *Divide the xy-Plane into Regions:* As a first step towards reducing the number of stored intervals, we divide the xy-plane into increasingly large intervals based on a user-supplied accuracy parameter $\epsilon > 0$. A smaller value of ϵ will indicate that the admission controller is more accurate (i.e., closer to the exact admission controller); however, the time complexity of the algorithm will be increased.

Definition 3 ($(1+\epsilon)$-Region): The i'th $(1+\epsilon)$-region denoted by $\mathcal{A}^i$ for $i \in \mathbb{N}^+$ is a horizontal strip in Euclidean space (i.e., $\mathbb{R}^2$) where the upper boundary is $(1+\epsilon)$ times the lower boundary of $\mathcal{A}^i$. Formally, the i'th $(1+\epsilon)$-region is defined as

$$\mathcal{A}^i \overset{\text{def}}{=} \left\{(x, y) \in \mathbb{R}^2 \mid (1+\epsilon)^{i-1} \le y < (1+\epsilon)^i\right\} \tag{7}$$

Figure 4 gives a visual depiction of the horizontal division of the xy-plane. We denote the lower bound and upper bound of region $\mathcal{A}^i$ as $\mathcal{A}^i.lb$ and $\mathcal{A}^i.ub$ respectively.

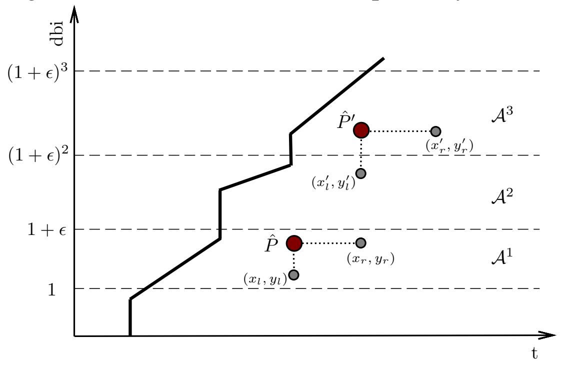


Fig. 4. Approximating Y-axes of dbi.

2) *Merge Intervals Within A Region:* Consider two distinct intervals represented by (x_l, y_l) and (x_r, y_r) in the same $(1+\epsilon)$-region $\mathcal{A}^i$. As a method of reducing the number of intervals stored, we will merge two such intervals into an *approximation point*.

Definition 4 (Approximation Point $\hat{P}((x_l, y_l), (x_r, y_r))$): Consider two points (x_l, y_l) and (x_r, y_r) where $x_l \le x_r$ and $y_l \le y_r$. We define the *approximation point* $\hat{P}((x_l, y_l), (x_r, y_r))$ "anchored" by points (x_l, y_l) and (x_r, y_r) as

$$\hat{P}((x_l, y_l), (x_r, y_r)) \overset{\text{def}}{=} (x_l, y_r). \tag{8}$$

The points (x_l, y_l) and (x_r, y_r) are referred to as the *left-anchor point* and *right-anchor point* of approximation point $\hat{P}$.

For simplicity, we drop the "$((x_l, y_l), (x_r, y_r))$" from $\hat{P}$ when it is clear which approximation point we are referring to. The notation $\hat{P}.x_l$ and $\hat{P}.y_l$ (respectively, $\hat{P}.x_r$ and $\hat{P}.y_r$) refers to the x and y coordinates of the left (right) anchor point. Figure 5(a) shows the formation of the approximation point $\hat{P}$ from its two anchor points. One important point to observe is that, due to the fact that we merge points in the same $(1+\epsilon)$-region, it must be that $\hat{P}.y_r \leq (1+\epsilon)\hat{P}.y_l$. In other words, the y-value of the approximation point is no more than a factor of $(1+\epsilon)$ greater than the y-value of its left anchor point. This observation will be useful in proving the approximation ratio of our admission controller. In the next step, we show how these approximation points can be utilized to decrease the total number of intervals stored by our approximate admission controller.

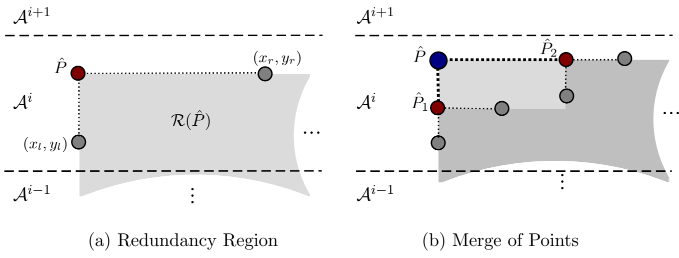


Fig. 5. Approximation point $\hat{P}$ in the $(1+\epsilon)$-region $\mathcal{A}^i$.

3) *Eliminate Redundant Points:* Observe from Figure 5(a) that the region below and to the right of the approximation point $\hat{P}$ forms a rectangular region extending infinitely to the right and below, called the *redundancy region* for approximation point $\hat{P}$.

Definition 5 (Redundancy Region): A *redundancy region* for approximation point $\hat{P}$ is the region extending towards lower right of a point in cartesian plane:

$$\mathcal{R}(\hat{P}) \stackrel{\text{def}}{=} \left\{ (x, y) \in \mathbb{R}^2 | (\hat{P}.x_l \leq x) \wedge (\hat{P}.y_r \geq y) \right\}. \quad (9)$$

The following observation allows us to ignore intervals corresponding to points that fall into this rectangular region; we call points falling into this region *redundant points*.

Lemma 4: For a given point $\hat{P}$ and any point $(x, y) \in \mathbb{R}^2$, if $x \geq \hat{P}.x_l$ and $y \leq \hat{P}.y_r$ then $\mathsf{dbi}(\Lambda, \hat{P}.x_l) - \hat{P}.y_r \leq \mathsf{dbi}(\Lambda, x) - y$.

Proof: Since dbi is a non-decreasing function, $\mathsf{dbi}(\Lambda, x) \geq \mathsf{dbi}(\Lambda, \hat{P}.x_l)$. Combining this with the condition $\hat{P}.y_r \geq y$, we obtain $\mathsf{dbi}(\Lambda, \hat{P}.x_l) - \hat{P}.y_r \leq \mathsf{dbi}(\Lambda, x) - y$. ■

Therefore, we can conclude that if approximate point $\hat{P}$ is below dbi, then any point in the redundancy region $\mathcal{R}(\hat{P})$ will also be below dbi.

4) *Merge Approximation Points:* In Step 2, we show how two intervals in the same $(1+\epsilon)$-region can be merged into a single approximation point. We will see in the next subsection that each approximation point is shifted to the right and upwards as new jobs are admitted to the system. Thus, an approximation point formed in region $\mathcal{A}^i$ may eventually move (completely with both anchor points) into another region $\mathcal{A}^j$ where $j > i$. We say that an approximation point $\hat{P}$ is "completely in" $\mathcal{A}^j$ if $\mathcal{A}^j.lb \leq \hat{P}.y_l \leq \hat{P}.y_r \leq \mathcal{A}^j.ub$; otherwise, the approximation point "straddles" $(1+\epsilon)$-regions. An approximation point $\hat{P}$ is "contained" (not necessarily completely) in $\mathcal{A}^j$, if $\mathcal{A}^j.lb \leq \hat{P}.y_r \leq \mathcal{A}^j.ub$ Consider any two approximation points $\hat{P}_1$ and $\hat{P}_2$ that are completely in $\mathcal{A}^j$; i.e., for $k \in \{1, 2\}$, the approximation points $\hat{P}_k$ satisfies $\hat{P}_k.y_l \geq \mathcal{A}^j.lb$ and $\hat{P}_k.y_r \leq \mathcal{A}^j.ub$. Given these two points, we may merge $\hat{P}_1$ and $\hat{P}_2$ to form a new approximation point $\hat{P} = \left(\min_{k\in\{1,2\}} \{\hat{P}_k.x_l\}, \max_{k\in\{1,2\}} \{\hat{P}_k.y_r\} \right)$. Figure 5(b) depicts the merge of two approximation points and evolution of their redundancy regions after merging.

A. Algorithm Description

In Figure 6, we present the pseudocode for our approximate admission control algorithm for an arbitrary demand-curve interface. The algorithm keeps a linked list of nodes L, where each node represents an approximation point corresponding to a $(1+\epsilon)$-region. Each node $\hat{P}$ consists of two points: left anchor (x_l, y_l) and right anchor (x_r, y_r), and a pointer next to the next approximation point. We abuse notation somewhat to allow $\hat{P}$ to refer to both the point in the xy-plane and the node in the list L; $\hat{P}.\mathsf{next}$ is the next node after $\hat{P}$ in the list or null if no such node exists. The nodes of L are ordered in increasing value of x_l. The list is initially empty. At any time the algorithm keeps a variable d_{last} to store the last admitted job's deadline.

First we describe some helper subroutines used by the approximate admission controller. The procedure ApproximateAC-Init() (Figure 6) initializes the list L to empty, d_{last} to zero and $\hat{P}_h$ to null. An Insert(x, y) operation (Figure 7) creates a node in the list L with the approximate point equal to the anchor points; i.e., $(x_l, y_l) = (x_r, y_r) = (x, y)$. Then the node is inserted into the list in non-decreasing order of its x_l-value. A Delete$(\hat{P})$ operation deletes node $\hat{P}$ from the list. A Shift$(\hat{P}, \delta_x, \delta_y)$ operation shifts both the left anchor and the right anchor of $\hat{P}$ by (δ_x, δ_y) amount, where δ_x is the shift in X-axes and δ_y is the shift in Y-axes. The Merge$(\hat{P}_1, \hat{P}_2)$ operation as described earlier in this section merges two nodes by updating the left anchor and right anchor of $\hat{P}_1$ with the left-most of the two left anchors, and the top-most of the two right anchors respectively (Figure 5(b)). Then it deletes the node $\hat{P}_2$ and updates the linked list accordingly.

When a new job $j_k(A_k, E_k, D_k)$ arrives in the system, ApproximateAC first checks in Line 1 whether the demand of the job E_k over the interval D_k exceeds the demand interface over a D_k-length interval, i.e., $\mathsf{dbi}(\Lambda, D_k)$. It is possible to show (similar to Lemma 2) that all the intervals of interest for MAD jobs end at the last admitted job's deadline d_{last}. Using this fact, the potential increase (due to accepting j_k) in the interval lengths is $\delta_x = d_k - d_{\text{last}}$ and the increase in demand is $\delta_y = E_k$. In the *while*-loop of Lines 7 to 15, we check that each of the approximate points will still fall below the $\mathsf{dbi}(\Lambda, \cdot)$ if they are shifted to the right by δ_x

```
ApproximateAC-Init()
1  ▷ Each node $\hat{P}$ in the list consists of two points $(x_l, y_l), (x_r, y_r)$,
2  ▷ and a pointer to node next. Initially the list is empty.
3  $\hat{P}_h \leftarrow$ null; $d_{\text{last}} \leftarrow 0$
```

```
ApproximateAC($\Lambda, j_k, \epsilon$)
1   if dbi($\Lambda, D_k$) < $E_k$
2       Reject.
3   $\delta_x \leftarrow d_k - d_{\text{last}}$; $\delta_y \leftarrow E_k$
    ▷ Insert node at the beginning of the list
4   $\hat{P}_c \leftarrow$ Insert($D_k - \delta_x, E_k - \delta_y$)
5   $\hat{P}_c$.next $\leftarrow \hat{P}_h$; $\hat{P}_h \leftarrow \hat{P}_c$
6   $\hat{P}_p \leftarrow \hat{P}_h$; $\hat{P}_c \leftarrow \hat{P}_h$.next
7   while $\hat{P}_c$ not null
8       if dbi($\Lambda, \hat{P}_c.x_l + \delta_x$) < $\hat{P}_c.y_r + \delta_y$
9           Delete the head of the list. Reject.
        ▷ $\hat{P}_p$ and $\hat{P}_c$ would have moved to same region
10      $j \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_p.y_l + \delta_y) \rceil$; $k \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_c.y_r + \delta_y) \rceil$
11      if $j == k$
12          if dbi($\Lambda, \min\{\hat{P}_p.x_l, \hat{P}_c.x_l\} + \delta_x$)
                < ($\max\{\hat{P}_p.y_r, \hat{P}_c.y_r\} + \delta_y$)
13              Delete the head of the list. Reject.
14      $\hat{P}_p \leftarrow \hat{P}_c$; $\hat{P}_c \leftarrow \hat{P}_c$.next
15  end while
16  Accept $j_k$.
17  Update($\hat{P}_h, \delta_x, \delta_y$)
18  $d_{\text{last}} \leftarrow d_k$
```

```
Update($\hat{P}_h, \delta_x, \delta_y$)
1   $\hat{P}_p \leftarrow \hat{P}_h$
2   shift($\hat{P}_p, \delta_x, \delta_y$)
3   $\hat{P}_c \leftarrow \hat{P}_p$.next
4   while $\hat{P}_c$ not null
5       shift($\hat{P}_c, \delta_x, \delta_y$)
6       $j \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_p.y_l) \rceil$; $k \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_c.y_r) \rceil$
        ▷ $\hat{P}_p$ and $\hat{P}_c$ have moved to same region
7       if $j == k$
8           Merge($\hat{P}_p, \hat{P}_c$)
9       $\hat{P}_p \leftarrow \hat{P}_c$; $\hat{P}_c \leftarrow \hat{P}_c$.next
10  end while
```

Fig. 6. Pseudo-code for approximate admission control of MAD jobs for an arbitrary demand-curve.

and upwards by δ_y. In particular, Line 8 determines if the individual points shifted will violate Λ; Line 12 determines if any new approximation point, formed by the shift (due to the merging described in Step 4 of the previous subsection), will violate the interface Λ. If a violation is detected, j_k will be rejected; otherwise, we may accept the job.

Once the conditions are verified for all the nodes in the list, the algorithm "accepts" the job and performs Update$(\hat{P}_h, \delta_x, \delta_y)$ operation for all nodes starting from the head of the list. This procedure shifts each node (both left and right anchors) by (δ_x, δ_y) amount using Shift and performs Merge operation when two consecutive nodes are in the same $(1+\epsilon)$-region.

```
Insert($x, y$)
1  Allocate a new node $\hat{P}$
2  $\hat{P}.x_l \leftarrow x$; $\hat{P}.x_r \leftarrow x$
3  $\hat{P}.y_l \leftarrow y$; $\hat{P}.y_r \leftarrow y$
4  $\hat{P}$.next $\leftarrow$ null
5  return $\hat{P}$
```

```
Delete($\hat{P}$)
1  Free the memory for node $\hat{P}$.
```

```
Shift($\hat{P}, x, y$)
1  $\hat{P}.x_l \leftarrow \hat{P}.x_l + x$; $\hat{P}.x_r \leftarrow \hat{P}.x_r + x$
2  $\hat{P}.y_l \leftarrow \hat{P}.y_l + y$; $\hat{P}.y_r \leftarrow \hat{P}.y_r + y$
```

```
Merge($\hat{P}_1, \hat{P}_2$)
1  if $\hat{P}_1.x_l > \hat{P}_2.x_l$
       ▷ Update left anchor
2      $\hat{P}_1.x_l = \hat{P}_2.x_l$; $\hat{P}_1.y_l = \hat{P}_2.y_l$
3  if $\hat{P}_1.y_r < \hat{P}_2.y_r$
       ▷ Update right anchor
4      $\hat{P}_1.x_r = \hat{P}_2.x_r$; $\hat{P}_1.y_r = \hat{P}_2.y_r$
5  $\hat{P}_1$.next = $\hat{P}_2$.next; Delete($\hat{P}_2$)
```

Fig. 7. Pseudo-code for approximate admission control of MAD jobs for an arbitrary demand-curve.

B. Proof of Correctness

We now show that our approximate admission controller is correct in the following theorem. Throughout this section, we consider a new element of L as an "approximation point" only after the admission controller ApproximateAC(Λ, j_k, ϵ) has returned from its execution.

Theorem 2: If ApproximateAC(Λ, j_k, ϵ) returns "Accept", then j_k may be admitted without violating Λ.

Our first lemma of this subsection show that we always "cover" any deleted point by another point; in other words, any deleted point must be contained in the redundancy region of a point that is in list L.

Lemma 5: For any approximation point $\hat{P}$ that was inserted into list L, if $\hat{P}$ is deleted, then there exists some approximation point $\hat{P}'$ in L such that $\hat{P}'.x_l \leq \hat{P}.x_l$ and $\hat{P}'.y_r \geq \hat{P}.y_r$.

Proof: A node is deleted only in Line 9 or 13 of ApproximateAC(Λ, j_k, ϵ) or in the Merge subroutine. Clearly, in both cases, this only deletes the node that was inserted in Line 4; thus, by the note above this lemma, we do not consider this to be an approximation point. Therefore, the only subroutine that can delete nodes from L is the Merge subroutine. Recall that when we merge approximation points $\hat{P}_1$ and $\hat{P}_2$, a new point $\hat{P}' = \left(\min_{k\in\{1,2\}}\{\hat{P}_k.x_l\}, \max_{k\in\{1,2\}}\{\hat{P}_k.y_r\}\right)$ is created. It is obvious that $\hat{P}_1$ and $\hat{P}_2$ are in the redundancy region $\mathcal{R}(\hat{P}')$ (refer to Figure 5(b)). Thus, for the two points deleted by the merge operation, by definition of redundancy region, $\hat{P}'.x_l \le \hat{P}_k.x_l$ and $\hat{P}'.y_r \ge \hat{P}_k.y_r$ for $k \in \{1, 2\}$. ■

The next lemma shows the correspondence between points stored in L and intervals ending at the deadline of last admitted job (i.e., d_{last}).

Lemma 6: After the call to APPROXIMATEAC-INIT() and the k'th invocation of APPROXIMATEAC(Λ, j_k, ϵ) where $k \in \mathbb{N}$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the approximate admission controller. For each job $j_\ell \in J_k$, there exists an approximation point $\hat{P}$ in list L such that $\hat{P}.x_l$ is at most $d_{\text{last}} - A_\ell$ and $\hat{P}.y_r$ is at least $\mathsf{demand}(J_k, A_\ell, d_{\text{last}})$. Furthermore, d_{last} equals $\max\{\{0\} \cup \{d_\ell \mid j_\ell \in J_k\}\}$.

Proof: We prove the lemma by induction on k.

Base Case: When $k = 0$, APPROXIMATEAC-INIT() has been invoked and thus no jobs have been admitted; i.e, $J_0 = \emptyset$. The lemma is clearly true as L is initialized to $\emptyset$ and d_{last} is initialized to zero.

Inductive Hypothesis: Assume that the lemma holds for each i ($i = 1, 2, \ldots, k-1$) successive calls to APPROXIMATEAC(Λ, j_i, ϵ).

Inductive Step: We must show that the lemma holds for the k'th call to APPROXIMATEAC(Λ, j_k, ϵ). The admission controller can either return "accept" or "reject". Let us first consider the case that APPROXIMATEAC(Λ, j_k, ϵ) returns "reject". Then, J_{k-1} is identical to J_k and d_{last} is not changed by any instruction in the execution path to "reject". Thus, by the inductive hypothesis, the lemma obviously continues to hold as the state is identical to after the call to APPROXIMATEAC(Λ, j_k, ϵ).

Now, consider the case when APPROXIMATEAC(Λ, j_k, ϵ) returns "accept". Line 3 of the procedure sets d_{last} equal to d_k; Let the updated value of d_{last} and L be denoted by $d_{\text{last}}^{\text{new}}$ and L^{new} respectively. Let $d_{\text{last}}^{\text{old}}$ and L^{old} denote the value of the d_{last} and L, prior to the invoke of APPROXIMATEAC(Λ, j_k, ϵ). By the inductive hypothesis, for each job $j_\ell \in J_{k-1}$, there exists $\hat{P} \in L^{\text{old}}$ such that $\hat{P}.x_l$ is at most $d_{\text{last}}^{\text{old}} - A_\ell$ and $\hat{P}.y_r$ is at least $\mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}}^{\text{old}})$. The SHIFT subroutine shifts each approximation point (and its anchors) to the right by $\delta_x = d_{\text{last}}^{\text{new}} - d_{\text{last}}^{\text{old}}$ and up by $\delta_y = E_k$. Lemma 5, implies that if any approximation point $\hat{P}$ is deleted by a MERGE, then L^{new} has an approximation point $\hat{P}'$ with $\hat{P}'.x_l \le \hat{P}.x_l$ and $\hat{P}.y_r \le \hat{P}'.y_r$. Thus, each $\hat{P} \in L^{\text{old}}$ is now $(\hat{P}.x_l + \delta_x, \hat{P}.y_r + \delta_y) \in L^{\text{new}}$ or has an approximation point $\hat{P}'$ that covers it. Furthermore, $\hat{P}.x_l + \delta_x$ is at most $(d_{\text{last}}^{\text{new}} - d_{\text{last}}^{\text{old}}) + d_{\text{last}}^{\text{old}} - A_\ell = d_{\text{last}}^{\text{new}} - A_\ell$ and $\hat{P}.y_r + \delta_y$ is at most $\mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}}^{\text{old}}) + E_\ell$. The last expression is equivalent to $\mathsf{demand}(J_k, A_\ell, d_{\text{last}}^{\text{new}})$ since increasing the interval length by δ_x includes only the new job j_k in the interval $[A_\ell, d_{\text{last}}^{\text{new}}]$. Finally, adding the point $\{(D_k - \delta_x, E_k - \delta_y)\}$ in Line 4 and shifting by δ_x and δ_y is equivalent to adding an approximation point $\hat{P} = (D_k, E_k)$ which equals $(d_{\text{last}}^{\text{new}} - A_k, \mathsf{demand}(J_k, A_k, d_{\text{last}}^{\text{new}}))$ to the list L. Thus, the lemma holds for J_k. ■

The next lemma shows that the approximate admission controller returns "Accept" for a job j_k only when it is safe to do so.

Lemma 7: After the call to APPROXIMATEAC-INIT() and the k'th invocation of APPROXIMATEAC(Λ, j_k, ϵ) where $k \in \mathbb{R}$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the approximate admission controller. It must be that Equation 2 holds for J_k and Λ.

Proof: We prove the lemma by induction on k.

Base Case: When $k = 0$, APPROXIMATEAC-INIT() has been invoked and thus no jobs have been admitted; i.e, $J_0 = \emptyset$. Clearly, $\mathsf{demand}(J_k, t_1, t_2)$ is zero for all valid choices of t_1 and t_2. Therefore, Equation 2 is trivially true.

Inductive Hypothesis: Assume that Equation 2 holds for each i ($i = 1, 2, \ldots, k-1$) successive calls to APPROXIMATEAC(Λ, j_i, ϵ). That is, J_i satisfies Equation 2.

Inductive Step: We must show that, given the inductive hypothesis, Equation 2 holds for J_k. During the call APPROXIMATEAC(Λ, j_k, ϵ), the admission control can either return "accept" or "reject". Let us first consider the case that APPROXIMATEAC(Λ, j_k, ϵ) returns "reject". Then, J_{k-1} is identical to J_k; since Equation 2 holds for J_{k-1}, by the inductive hypothesis, it obviously continues to hold for J_k.

Now consider the case that APPROXIMATEAC(Λ, j_k, ϵ) returns "accept". Lemma 6 implies that for each $j_\ell \in J_k$ there is an approximation point $\hat{P}$ with $\hat{P}.x_l \le d_k - A_\ell$ and $\hat{P}.y_r \ge \mathsf{demand}(J_k, A_\ell, d_k)$. The fact that the j_k was accepted implies Line 8 was satisfied for each approximation point; that is,

$$\forall j_\ell \in J_k, \mathsf{dbi}(\Lambda, d_k - A_\ell) \ge \mathsf{demand}(J_k, A_\ell, d_k). \qquad (10)$$

The inductive hypothesis implies that prior to the invoke of APPROXIMATEAC(Λ, j_k, ϵ), Equation 2 held; therefore,

$$\begin{aligned} &\forall j_i, j_j \in J_{k-1} : d_i \le d_j :: \\ &\mathsf{demand}(J_{k-1}, A_i, d_j) \le \mathsf{dbi}(\Lambda, d_j - A_i). \end{aligned} \qquad (11)$$

Since d_k is later than any other d_i in J_{k-1}, j_k does not increase the demand in the above term $\mathsf{demand}(J_{k-1}, A_i, d_j)$. Thus, the condition still holds when the left-hand side of the inequality is replaced with $\mathsf{demand}(J_k, A_i, d_j)$. This observation regarding Equation 11 along with Equation 10 satisfy the supposition of Lemma 1 for the job set J_k; therefore, Equation 2 is satisfied for J_k. ■

Theorem 2 immediately follows from the above lemma.

C. Proof of Approximation Ratio

In this section, we argue that when the approximate admission controller rejects a job, then the exact admission controller would have also done so on a slightly "smaller" dbi. The accuracy of the test is determined by the accuracy parameter $\epsilon > 0$. While not required for the correctness of the algorithm, to prove the approximation ratio, we do require that each job j_i have execution time E_i at least equal to one. (The algorithm will still work correctly for $E_i < 1$; however, the approximation ratio is not true in this case).

Our first lemma of the subsection shows that the ratio of y-values of the right and left anchor point is bounded in terms of ϵ.

Lemma 8: For any approximation points $\hat{P}$ the following invariant holds:

$$\hat{P}.y_r \leq (1+\epsilon)\hat{P}.y_l. \tag{12}$$

Proof: Observe the only operations that change an approximation point are INSERT, SHIFT, and MERGE. We will show that, if the invariant initially holds for a point, the invariant will continue to hold after each operation. The INSERT operation creates a new approximation point with left anchor point equal to the right anchor point; thus, the invariant of the lemma initially holds. Now consider the SHIFT operation applied to an approximation point $\hat{P}$ where the invariant holds. The SHIFT operation is only called from Lines 2 and 5 of UPDATE. Let $\hat{P}'$ represent the approximation point after the application of $\text{SHIFT}(\hat{P}, \delta_x, \delta_y)$; thus, $\hat{P}'.y_r = \hat{P}.y_r + \delta_y$ and $\hat{P}'.y_l = \hat{P}.y_l + \delta_y$. If the invariant is true prior to the SHIFT call, then $\hat{P}'.y_r = \hat{P}.y_r + \delta_y \leq (1+\epsilon)\hat{P}.y_l + \delta_y \leq (1+\epsilon)(\hat{P}.y_l + \delta_y) = (1+\epsilon)\hat{P}'.y_l$. Thus, the invariant continues to hold after SHIFT.

Finally, to see that the invariant holds after MERGE, observe that the algorithm only merges two consecutive points $\hat{P}_\ell$ and $\hat{P}_{\ell+1}$ in the list L when both anchor points of these approximation points are completely within some region $\mathcal{A}^i$. For consecutive points $\hat{P}_\ell$ and $\hat{P}_{\ell+1}$ in L, it may be shown (Lemma 11 in Section V-D) that $\hat{P}_\ell.y_r \leq \hat{P}_{\ell+1}.y_r$. This observation taken together with the fact that the points are completely within $\mathcal{A}^i$ implies that $\mathcal{A}^i.lb \leq \hat{P}_\ell.y_l \leq \hat{P}_\ell.y_r \leq \hat{P}_{\ell+1}.y_r \leq \mathcal{A}^i.ub = (1+\epsilon)\mathcal{A}^i.lb$; the last inequality follows from Definition 3. Since $\mathcal{A}^i.lb \leq \hat{P}_\ell.y_l$, it must be that $\hat{P}_{\ell+1}.y_r \leq (1+\epsilon)\mathcal{A}^i.lb \leq (1+\epsilon)\hat{P}_\ell.y_l$. The MERGE operation will make a new approximation point $\hat{P}'$ with $\hat{P}'.y_l = \hat{P}_\ell.y_l$ and $\hat{P}'.y_r = \hat{P}_{\ell+1}.y_r$; thus, the invariant continues to hold for $\hat{P}'$. ■

The following corollary follows from the observation that a left anchor point below $\mathcal{A}^{i-1}.lb$ and a right anchor point above $\mathcal{A}^{i-1}.ub$ (equal to $\mathcal{A}^i.lb$) would violate the invariant of Lemma 8.

Corollary 1: For an approximation point $\hat{P}$ with $\mathcal{A}^i.lb \leq \hat{P}.y_r \leq \mathcal{A}^i.ub$ and $\hat{P}.y_l < \mathcal{A}^i.lb$ where $i > 1$, the left anchor point of $\hat{P}$ must be in the $(1+\epsilon)$-region $\mathcal{A}^{i-1}$; that is, $\mathcal{A}^{i-1}.lb \leq \hat{P}.y_l \leq \mathcal{A}^{i-1}.ub$.

Proof: Lets denote the left anchor as P_L and right anchor as P_R. Given $P_L \in \mathcal{A}^i$ and $P_L \in \mathcal{A}^j$ where $j < i$, we prove the proposition by contradiction. Assume $j = i-2$, that is, left anchor P_L is in region $\mathcal{A}^{i-2}$. Let q denote the distance of the anchor points in y-axes, that is $q = \hat{P}.yr - \hat{P}.yl$. By our assumption, q must be greater then the width of the region $\mathcal{A}^{i-1}$, i.e., $q > (1+\epsilon)^{i-2}$. Since approximation points with anchors in different regions are only created by shifting the points (upward and right), $\hat{P}$ must have been formed as an approximation point in any region $\mathcal{A}^\ell$, where $\ell \leq i-2$ (as $\hat{P}_L \in \mathcal{A}^{i-2}$). By definition of $(1+\epsilon)$-regions, the width of any such region is less or equal $(1+\epsilon)^{i-3}$. When an approximation point is created, both the anchors are within the same region, thus, the distance q in y-axes between $\hat{P}_L$ and $\hat{P}_R$ must be less or equal the width of region $\mathcal{A}^{i-2}$. This contradicts our assumption, since the relative distance between the anchors stays same after the shift operation which moved $\hat{P}_R$ to $\mathcal{A}^i$, and no merge is performed when anchors are in different regions . Therefore, we can conclude that $\hat{P}_L$ must reside in region $\mathcal{A}^{i-1}$. ■

The next lemma shows that for any approximation point, the left anchor corresponds to the exact demand over some interval.

Lemma 9: After the call to APPROXIMATEAC-INIT() and the k'th invocation of $\text{APPROXIMATEAC}(\Lambda, j_k, \epsilon)$ where $k \in \mathbb{N}$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the approximate admission controller. For each $\hat{P}$ in L, it must be that there exists a $j_\ell \in J_k$ such that $\mathsf{demand}(J, A_\ell, d_{\text{last}})$ equals $\hat{P}.y_l$ and $d_{\text{last}} - A_\ell$ equals $\hat{P}.x_l$.

Proof: We prove the lemma by induction on k.

Base Case: When $k = 0$, APPROXIMATEAC-INIT() has been invoked and thus no jobs have been admitted; i.e, $J_0 = \emptyset$. Clearly, L is empty and the lemma is vacuously true.

Inductive Hypothesis: Assume after i ($i = 1, 2, \ldots, k-1$) successive calls to $\text{APPROXIMATEAC}(\Lambda, j_i, \epsilon)$ there is a corresponding exact interval for J_i for each left anchor point $\hat{P} \in L$.

Inductive Step: If $\text{APPROXIMATEAC}(\Lambda, j_k, \epsilon)$ reject j_k, then the points of L are unaffected and J_{k-1} equals J_k; thus, by the inductive hypothesis, the lemma holds. If $\text{APPROXIMATEAC}(\Lambda, j_k, \epsilon)$ accepts j_k, then we must argue that each approximation point continues to have its left anchor point correspond to the exact demand of J_k over some interval. The only subroutines that could affect the approximation points of list L are INSERT, UPDATE, and MERGE. For INSERT, the newly-inserted point corresponds to the demand of job j_k. For UPDATE each point (including anchors) is shifted upwards $\delta_y = E_k$ and to the right by $\delta_x = d_k - d_{\text{last}}$. By the inductive hypothesis, for any $\hat{P} \in L$, there exists $j_\ell \in J_{k-1}$ such that $\hat{P}.x_l = d_{\text{last}} - A_\ell$ and $\hat{P}.y_l = \mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}})$. Let $\hat{P}' \in L$ be the resulting point after the SHIFT; thus, $\hat{P}'.x_l = d_{\text{last}} - A_\ell + d_k - d_{\text{last}} = d_k - A_\ell$ and $\hat{P}'.y_l = \mathsf{demand}(J_{k-1}, A_\ell, d_{\text{last}}) + E_k$ which equals $\mathsf{demand}(J_k, A_\ell, d_k)$; furthermore, d_{last} gets updated to d_k. Thus, the lemma holds after a SHIFT. For MERGE, by inductive hypothesis and the fact the lemma holds after INSERT and SHIFT operations, the left anchors of the two points merged together correspond to the exact demand of J_k over some interval. By definition of merging, one of the left anchor points of the merged points becomes the new left anchor of the new approximation point. Thus, the left anchor point continues to correspond to the exact demand over the same interval. ■

We may now quantify the inaccuracy of our approximate admission controller by proving the following theorem.

Theorem 3: Given a set of previously-admitted jobs J, if ApproximateAC(Λ, j_k, ϵ) returns "Reject", then ExactAC(Λ, j_k) also returns "Reject" for a demand-curve $\frac{1}{1+\epsilon}\mathsf{dbi}(\Lambda, \cdot)$ on the same previously-admitted job set.

Proof: If ApproximateAC(Λ, j_k, ϵ) returns "Reject" given previously admitted job set J, then there exists an approximation point $\hat{P}$ that fails the test in either Line 1 (i.e., the execution of j_k is too large over it's arrival to deadline interval), Line 8 (i.e., it would fail after the Shift operation is applied), or Line 12 (i.e., it would fail after being merged with another point in the same $(1+\epsilon)$-region). Let us first assume that $\hat{P}$ would fail the test of Line 1; clearly j_k would fail the test of Line 4 of ExactAC.

By Lemma 9, $\hat{P}.x_l$ equals $d_{\text{last}} - A_\ell$ and $\hat{P}.y_l$ equals $\mathsf{demand}(J, A_\ell, d_{\text{last}})$ for some $J_\ell \in J$. If $\hat{P}$ fails in Line 8, then $\mathsf{dbi}(\Lambda, \hat{P}.x_l + \delta_x) < \hat{P}.y_r + \delta_y$ which implies that $\mathsf{dbi}(\Lambda, d_k - A_\ell) < \hat{P}.y_r + \delta_y \leq (1+\epsilon)\hat{P}.y_l + \delta_y = (1+\epsilon)\mathsf{demand}(J, A_\ell, d_{\text{last}}) + E_k \leq (1+\epsilon)\mathsf{demand}(J \cup \{j_k\}, A_\ell, d_k)$ by Lemmas 8 and 9. We may similarly show that the lemma holds if Line 12 fails. Thus, the approximation ratio holds in all cases. ■

D. Algorithm Complexity

The complexity of the admission controller depends on the size of the linked list, since both ApproximateAC and Update have a loop traversing the list from beginning to end. We argue that the number of approximation points(nodes) for each $(1+\epsilon)$-region is constant (at most two). Therefore, the complexity of the algorithm is directly proportional to the number of $(1+\epsilon)$-regions. Our first lemma shows that each newly-admitted job corresponds to a node at the front of list L.

Theorem 4: For $\forall i$, a $(1+\epsilon)$-region $\mathcal{A}^i$ contains at most two approximation points.

Lemma 10: After the call to ApproximateAC-Init() and the k'th invocation of ApproximateAC(Λ, j_k, ϵ) where $k \in \mathbb{N}$ and $\epsilon > 0$, for MAD-sequenced jobs $j_1, j_2, \ldots, j_k$, define J_k to be the set of jobs admitted by the approximate admission controller. The approximation point $\hat{P} = (D_k, E_k)$ is the first node of list L; i.e., the newly-admitted job is at the front of the list.

Proof: A node corresponding to a newly-arrived job j_k arriving in MAD order will have the smallest interval length. (Recall that all existing intervals to be checked ending at d_k. Due to MAD ordering property, $A_k \geq A_\ell$ for all $j_\ell \in J_k$, thus $[A_k, d_k]$ is clearly the smallest interval). Since L is ordered in increasing x, a newly-inserted approximation point will always be inserted as a node at the front of list L. ■

Based on this observation, the following lemma states that for any two approximation point $\hat{P}_1$ and $\hat{P}_2$ the vertical region between the anchor points of the two points does not overlap. In other words, the anchor points are totally ordered with respect to their y-values.

Lemma 11: For any two approximation points $\hat{P}_1$ and $\hat{P}_2$ in list L where $\hat{P}_1$ precedes $\hat{P}_2$, the following invariant holds:

$$\hat{P}_1.y_l \leq \hat{P}_1.y_r \leq \hat{P}_2.y_l \leq \hat{P}_2.y_r. \qquad (13)$$

Proof: Observe that the operations that change an approximation point are Insert, Shift, and Merge. We will show that, if the invariant initially holds for a point, the invariant will continue to hold after each of the operations. Obviously, the invariant holds when we have an initially empty list. Let us first consider the Insert operation executed during the acceptance of some job j_k. By Lemma 10, we observed that a new node is inserted at the beginning of the list; furthermore, all subsequent approximation points (and their respective anchor points) are shifted upwards by $\delta_y = E_k$. Thus, since the newly created approximation point has a y-value of E_k for both anchor points and the point is entirely below the y-value of any other approximation point's anchor. The invariant will hold for the new point and every other point.

Now consider the Shift operation for any two approximation points $\hat{P}_1$ and $\hat{P}_2$. The Shift operation moves each anchor point of $\hat{P}_1$ and $\hat{P}_2$ upwards by the same amount; thus, the invariant continues to hold for $\hat{P}_1$ and $\hat{P}_2$ after the shift operation is applied to each approximation point in the list.

Finally, for the Merge operation, consider two successive points $\hat{P}$ and $\hat{P}'$ in list L that are merged together to create $\hat{P}_1$. Consider a third point $\hat{P}_2$. If $\hat{P}_2$ appears later in the list, then prior to the call to Merge (under the assumption that the invariant holds) we had $\hat{P}.y_l \leq \hat{P}.y_r \leq \hat{P}'.y_l \leq \hat{P}'.y_r \leq \hat{P}_2.y_l \leq \hat{P}_2.y_r$. After the call to Merge, the approximation point $\hat{P}_1$ has left anchor point $(\hat{P}.x_l, \hat{P}.y_l)$ and right anchor point $(\hat{P}'.x_r, \hat{P}'.y_r)$. Thus, the invariant continues to hold for this case. The lemma may be shown symmetrically if P_2 precedes $\hat{P}$ and $\hat{P}'$. ■

Using this lemma, we will prove that the list will have at most two nodes corresponding to a $(1+\epsilon)$-region.

Lemma 12: For $\forall i$, $(1+\epsilon)$-region $\mathcal{A}^i$ contains at most one approximation point such that $(\mathcal{A}^i.lb \leq \hat{P}.y_r \leq \mathcal{A}^i.ub) \wedge (\mathcal{A}^i.lb \leq \hat{P}.y_l \leq \mathcal{A}^i.ub)$ after each call to ApproximateAC.

Proof: The nodes in L are ordered in non-decreasing y-value of right anchors by Lemma 11. The Insert operation inserts new node at the beginning of the list by Lemma 10, the Shift operation shifts all the nodes same amount in X and Y-axes, and finally the Merge operation merges consecutive nodes that are in same region. Clearly, Merge will eliminate all but one approximation point for a $(1+\epsilon)$-region that may have temporarily contained more than one point. ■

The next corollary shows that at most one approximation point may "straddle" any $(1+\epsilon)$-region boundary. In other words, at most one approximation point has left anchor point below and right anchor point above the boundary $\mathcal{A}^i.lb$ for any $i > 1$. This corollary follows by the observation that if there were two such approximation points, then they would have to overlap in terms of y-value which would contradict the invariant of Equation 13 of Lemma 11.

Corollary 2: For $\forall i$, region $\mathcal{A}^i$ contains at most one approximation point such that $(\mathcal{A}^i.lb \leq \hat{P}.y_r \leq \mathcal{A}^i.ub) \wedge (\hat{P}.y_l \leq$

$\mathcal{A}^i.lb$).

Combining Lemma 12 and Corollary 2, we see that at most one approximation point exist completely inside any region $\mathcal{A}^i$ and at most one straddles the boundary of $\mathcal{A}^i.lb$ for $i \geq 1$. This proves Theorem 4, which upper bounds the number of intervals that the approximate admission controller tracks.

Despite bounding the number of approximation points per $(1+\epsilon)$-region, the number of these regions could be potentially infinite, for an arbitrary dbi; however, practically, a system cannot define an arbitrary infinite dbi function. Typically, there are two design choices: 1) store the dbi as a finite set of linear segments (i.e., the dbi has a finite number of steps and an entry for each step in the function); or 2) dbi is generated from some finite set of recurring tasks and each point can be calculated using some known closed-form equation. In [22], we explore the second option by showing how to modify the algorithm to handle a dbi generated from periodic/sporadic task systems. Thus, for now, we will assume that we are given a finite number of line segments as dbi.

Let r be the number of line segments required to specify the dbi for Λ. We can view the dbi as an ordered set $\{((a_i, b_i), s_i) \mid 1 \leq i \leq r\}$ where elements are ordered in increasing a_i values and (a_i, b_i) represents the left endpoint of the i'th line segment and s_i is the slope of the line segment. In other words, for any interval length $t \in [a_i, a_{i+1})$ for some $i : 1 \leq i < r$ the $\mathsf{dbi}(\Lambda, t) = b_i + (t - a_i)s_i$. Furthermore, to ensure that dbi is non-decreasing, $b_i + (a_{i+1} - a_i)s_i \leq b_{i+1}$ for all $i : 1 \leq i < r$. For approximation points $\hat{P}.x_l$ larger than a_r, we may modify APPROXIMATEAC to use a constant-time approach given in Dewan and Fisher [20] for doing admission control of a single-step dbi by storing only a single point with $\hat{P}.x_l \geq a_r$. Otherwise, let $\mathcal{U}$ equal $b_{r-1} + (a_r - a_{r-1})s_{r-1}$. For all points before a_r, APPROXIMATEAC requires at most $\lceil \log_{1+\epsilon} \mathcal{U} \rceil$ different $(1+\epsilon)$-regions to cover all the approximation points. Thus, we have at most $O(\log_{1+\epsilon} \mathcal{U})$ approximation points. To calculate the dbi at any of the approximation points, we must simply look up the value in the ordered list which may be accomplished in $O(\log r)$ time complexity. Thus, the overall complexity of the approach for a finite stored dbi is $O((\log_{1+\epsilon} \mathcal{U})(\log r))$. The value of $\mathcal{U}$ is at most exponential (with base of two) in the number of bits to represent the line segments of the dbi. Our approximate admission controller therefore has complexity that is polynomial in the number of bits to store each line segment and $1/\epsilon$. Furthermore, the worst-case computational complexity does not depend on the number of jobs admitted in during the lifetime of the system; this removes a fundamental drawback of the exact admission controller.

We derive an upper bound on periodic dbi in Section VI. Let the upper bound for the dbi is $\mathcal{U}$, then the number of regions will be $\log_{1+\epsilon} \mathcal{U}$ by Definition 12. Since all other operations take constant time, the complexity of the algorithm is $O(\log_{1+\epsilon} \mathcal{U})$.

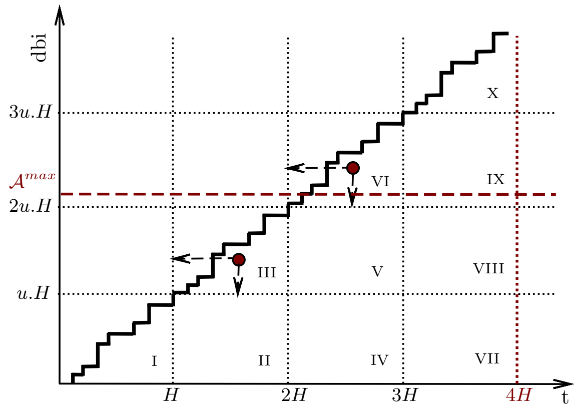


Fig. 8. Deriving Upper Bound for Periodic dbi.

VI. DERIVING UPPER-BOUND FOR dbi

In this section, we consider the second possibility posed in Section V-D: the demand-curve interface is a periodic function. In particular, we assume that we are given a demand-curve generated from n constrained-deadline sporadic tasks [23]: $\tau \overset{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is specified by the tuple (e_i, d_i, p_i) where e_i is the worst-case execution requirement, d_i is the relative deadline, and p_i is the period of the task; we further assume that $d_i \leq p_i$ for all τ_i. Let u denote the total utilization of τ, and equal to $\sum_{i=1}^{n} e_i/p_i$. It has been shown [24] that the demand-curve $\mathsf{dbi}(\tau, t)$ equals $\sum_{\tau_i \in \tau} \lfloor (t - d_i)/p_i + 1 \rfloor \cdot e_i$. Furthermore, it can be easily shown that this dbi repeats with *hyperperiod* H equal to $\mathsf{lcm}_{i=1}^{n}\{p_i\}$; this repetition implies

$$\forall 0 \leq t \leq H :: \mathsf{dbi}(\tau, t + H) = \mathsf{dbi}(\tau, t) + uH. \quad (14)$$

Figure 8 shows the periodic behavior of the dbi. Note in the figure, we have divided the xy-plane into rectangular regions labeled with roman numerals. The width of each region is H and the height is uH. Due to the periodic nature of the dbi, Regions I, III, VI, X, etc. are identical; similarly, the Regions II, V, IX, etc. are also identical. Clearly a point (x, y) in any region with $x \geq H$ and $y \geq uH$ can be "mapped" to a point in its identical region (i.e., one region down and one region to the left) by $(x - H, y - uH)$. From Equation 14, we may observe that

$$\mathsf{dbi}(\tau, x - H) \geq y - uH \Leftrightarrow \mathsf{dbi}(\tau, x) \geq y. \quad (15)$$

Given the above observations regarding $\mathsf{dbi}(\tau, t)$, we now present a modification of the approximate admission controller to limit the number of regions to polynomial in ϵ and the size of τ. Let $\mathcal{A}^{\max}$ be the first $(1 + \epsilon)$-region completely above $2uH$; the region $\mathcal{A}^{\max}$ represents the last region that we will track. When an approximation point $\hat{P}$ crosses $\mathcal{A}^{\max}.lb$ (i.e., $\hat{P}.y_l$ crosses this boundary). We check whether the point entered via Region VI or IX; if so, we use the mapping of Equation 15 to map the point back to Region III or V, respectively. Otherwise, if it crossed $\mathcal{A}^{\max}.lb$ in some other region, we may show (in the extended version of this paper) that if this point ever violates the dbi, there exists

```
dbi-WrapCheck($\Lambda, \hat{P}, \epsilon$)
1   ▷ Let $A^{max}$ be the first region above $2uH$.
2   if $\hat{P}.x_l > 4H$
3         Delete all the nodes from $\hat{P}$ to the end of the list.
4   if $\hat{P}.y_l > A^{max}.lb$
5         $\hat{P}.x_l \leftarrow \hat{P}.x_l - H$
6         $\hat{P}.y_r \leftarrow \hat{P}.y_r - uH$
7         $r \leftarrow \lceil \log_{1+\epsilon}(\hat{P}.y_l) \rceil$
8         if $\mathcal{A}^r$ has approximation point ($\hat{P}_1$)
9               Merge($\hat{P}_1, \hat{P}$)
10        else
11              Insert($x_l, y_r$)
```

Fig. 9. Pseudo-code for checking dbi upper bounds.

a smaller interval that also does. Therefore, we can remove this point from consideration. We test for this by checking the condition $\hat{P}.x_l > 4H$. Figure 9 shows this procedure called dbi- WrapCheck which we may add in the Update subroutine in ApproximateAC after the Shift of each node. We can show that the number of $(1+\epsilon)$-regions for a periodic dbi is $O(\log_{1+\epsilon} uH)$ based the definition of $\mathcal{A}^{\max}$. Since $u \le n$ and $H \le p_{\max}^n$ where $p_{max} = \max_{\tau_i}\{p_i\}$, then the total number of regions can be upper-bounded by $O(n \log_{1+\epsilon} p_{\max})$. The time to compute the periodic dbi for any interval length adds an additional factor of $O(n)$ to the time complexity.

A. Eliminating Approximation Points for Periodic dbi

Finally, we show that we may discard an approximation point $\hat{P}$ if it is "too far away" from $\mathsf{dbi}(\Lambda, \cdot)$ for a dbi generated from a set of periodic tasks. Again, let H be the hyperperiod and u be the utilization of the task system. The first lemma gives a technical result regarding periodic dbis.

Lemma 13: For any positive real numbers x and y, it must be that

$$\mathsf{dbi}(\Lambda, x+y) - \mathsf{dbi}(\Lambda, y) \ge \mathsf{dbi}(\Lambda, x). \tag{16}$$

Proof: We use the following property of floors that for any a and b: $\lfloor a \rfloor + \lfloor b \rfloor \le \lfloor a+b \rfloor$. By $d_i \le p_i$ and the definition of dbi for sporadic tasks, we have

$$\begin{aligned} &\textstyle\sum_{i=1}^n \left(\left\lfloor \frac{x+y-d_i}{p_i} \right\rfloor + 1 \right) e_i - \sum_{i=1}^n \left(\left\lfloor \frac{y-d_i}{p_i} \right\rfloor + 1 \right) e_i \\ &= \textstyle\sum_{i=1}^n \left(\left\lfloor \frac{x+y-d_i}{p_i} \right\rfloor - \left\lfloor \frac{y-d_i}{p_i} \right\rfloor \right) e_i. \end{aligned} \tag{17}$$

Letting $a = \frac{x}{p_i}$ and $b = \frac{y-d_i}{p_i}$ and applying the property of floors, implies the lemma. ■

The next lemma shows that if the left anchor point of an approximation point $\hat{P}$ passes the boundary of $4H$ and its y-value less than $2uH$, then there exists another point (with smaller x-value) that will "cover" $\hat{P}$.

Lemma 14: Let $\hat{P}$ be an approximation point such that $\hat{P}.x_l > 4H$ and $\hat{P}.y_r \le 2uH$ at some time $t > 0$ and for any sequence of admitted jobs after t, $\hat{P}$ is not merged with any other point. Let $\hat{P}^{\text{new}}$ be the approximation point corresponding to $\hat{P}$ after jobs arriving between t and t' have been admitted where $t < t'$. If $\hat{P}^{\text{new}}.y_r > \mathsf{dbi}(\Lambda, \hat{P}^{\text{new}}.x_l)$ at time t', then there exists $\hat{P}' \in L$ at time t' such that $\hat{P}'.x_l \le \hat{P}.x_l$ and $\hat{P}'.y_r > \mathsf{dbi}(\Lambda, \hat{P}'.x_l)$.

Proof: Let J be the set of jobs admitted by ApproximateAC before t. Let J' be the set of jobs admitted after t up until t'. Observe by supposition of the lemma, $\hat{P}.x_l > 4H$ which implies that $\mathsf{dbi}(\Lambda, \hat{P}.x_l) > \mathsf{dbi}(\Lambda, 4H) = 4uH$. Since $\hat{P}.y_r \le 2uH$, it must be that

$$\mathsf{dbi}(\Lambda, \hat{P}.x_l) - \hat{P}.y_r \ge 2uH. \tag{18}$$

Since $\hat{P}$ is not merged, only Shift operations will change the location of $\hat{P}$. Let $J = \{j_1, j_2, \ldots, j_k\}$ and $J' = \{j_{k+1}, \ldots, j_n\}$. We denote the value of the variable d_{last} at time t as $d_{\text{last}}^{\text{old}}$; similarly, $d_{\text{last}}^{\text{new}}$ represents the variable at time t'. Please note by our definition of J and J', it must be that $d_{\text{last}}^{\text{old}} = d_k$ and $d_{\text{last}}^{\text{new}} = d_n$. By Lemma 6, there exists $j_i \in J$ such that $\hat{P}.x_l \le d_{\text{last}}^{\text{old}} - A_i = d_k - A_i$ and $\hat{P}.y_r \ge \mathsf{demand}(J, A_i, d_k)$. Since j_k was accepted $\hat{P}.y_r \le \mathsf{dbi}(\Lambda, \hat{P}.x_l) \le \mathsf{dbi}(\Lambda, d_k - A_i)$.

Let $\bar{\delta}_x$ equal $d_{\text{last}}^{\text{new}} - d_{\text{last}}^{\text{old}}$. Let $\bar{\delta}_y$ equals the total execution of all jobs of J'. The approximation point $\hat{P}^{\text{new}}$ is the original approximation point $\hat{P}$ shifted to the right by $\bar{\delta}_x$ and upwards by $\bar{\delta}_y$. If $\hat{P}^{\text{new}}.y_r > \mathsf{dbi}(\Lambda, \hat{P}^{\text{new}}.x_l)$ is true, then

$$\hat{P}.y_r + \bar{\delta}_y > \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x). \tag{19}$$

Combining Equations 18 and 19, we obtain

$$\bar{\delta}_y > \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x) - \mathsf{dbi}(\Lambda, \hat{P}.x_l) + 2uH. \tag{20}$$

Now consider the interval $[A_{k+1}, d_n]$. Observe that $d_n - A_i = (d_n - A_{k+1}) + (A_{k+1} - A_i)$. By Lemma 13,

$$\begin{aligned} &\mathsf{dbi}(\Lambda, d_n - A_{k+1}) \\ &\quad \le \mathsf{dbi}(\Lambda, d_n - A_i) - \mathsf{dbi}(\Lambda, A_{k+1} - A_i) \\ &\quad \le \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x) - \mathsf{dbi}(\Lambda, A_{k+1} - A_i) \\ &\quad \le \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x) - \Big[\mathsf{dbi}(\Lambda, \hat{P}.x_l) - uH \Big] \end{aligned} \tag{21}$$

The last inequality follows since $d_{k+1} \ge d_k$ (due to the MAD property). Assuming that $D_{k+1} \le H$, $\mathsf{dbi}(\Lambda, A_{k+1} - A_i) \ge \mathsf{dbi}(\Lambda, (d_k - H) - A_i) = \mathsf{dbi}(\Lambda, d_k - A_i) - uH \ge \mathsf{dbi}(\Lambda, \hat{P}.x_l) - uH$.

By Lemmas 5 and 6, there exist an approximation point $\hat{P}' \in L$ at time t' such that $\hat{P}'.x_l \le d_n - A_{k+1}$ and $\hat{P}'.y_r \ge \mathsf{demand}(J \cup J', A_{k+1}, d_n) = \bar{\delta}_y$. Combining these facts with Equations 18 and 21, we obtain

$$\hat{P}'.y_r > \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x) - \mathsf{dbi}(\Lambda, \hat{P}.x_l) + 2uH \tag{22}$$

and

$$\mathsf{dbi}(\Lambda, \hat{P}'.x_l) \le \mathsf{dbi}(\Lambda, \hat{P}.x_l + \bar{\delta}_x) - \mathsf{dbi}(\Lambda, \hat{P}.x_l) + uH. \tag{23}$$

Clearly, Equations 22 and 23 implies that $\hat{P}'.y_r > \mathsf{dbi}(\Lambda, \hat{P}'.x_l)$; if $\hat{P}'.x_l \le 4H$, then the lemma is proved. If not, we can repeat the same logic above to find $\hat{P}''$ with

$\hat{P}''.y_r > \mathsf{dbi}(\Lambda, \hat{P}''.x_l)$ where $\hat{P}''.x_l < \hat{P}'.x_l$. ■

B. Server Design

Our admission controllers ensure that the total system demand for the admitted jobs will never violate the demand-curve interface for the subsystem by policing the jobs before executing them. However, we need a mechanism to strictly enforce the interface at runtime; for example, if a job needs to execute more than its worst-case execution time, the system must ensure that temporal isolation is still maintained. We denote the worst-case execution time E_i specified in our aperiodic job model as the *estimated execution time* and $\bar{E}_i$ as the *actual execution time* of job j_i. In this section we address how the overrun ($E_i < \bar{E}_i$) and underrun ($E_i > \bar{E}_i$) situations can be handled so that the interface is not violated. We explore the design of a lightweight server to enforce temporal isolation and reclaim any unused reservation from a job that finishes early.

1) Temporal Isolation: Each job will have a server associated with it, which will monitor the execution time of the job. If the actual execution goes beyond the estimated execution time, the server will stop running the job. Upon j_i's admittance to the system (via our admission controller), the server is given a budget equal to E_i. The server consumes this budget at a linear rate only when executing the job. When the budget of the server for j_i is depleted, its execution is halted. If j_i has not successfully completed at this point, we have an overrun situation. The subsystem designer needs to determine whether to abort the job or spawn a new server by using the admission control algorithm with a new estimate of the remaining execution. Note that since the server does not permit the actual execution $\bar{E}_i$ of a job to exceed E_i, it enforces strong temporal isolation between subsystems.

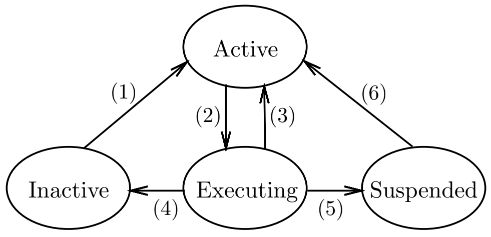


Fig. 10. State transition diagram.

Each server assigns a *state* to the admitted job. A job can be in one of the four states: *inactive*, *active*, *executing* and *suspended*. An inactive state is initial state of a job that just arrived to the system. Once the admission controller admits the job, it moves to active state (transition (1)). The job moves to executing state when the scheduler starts executing the job (transition (2)), and goes back to active state when a higher priority job preempts the execution of the job (transition (3)). While in executing state, if the total execution of the job exceeds E_i, the server transitions the job to suspended state (5). The job goes back to inactive state when it finishes execution (transition 4). From the suspended state, the job moves back to active state (6) when the job with new execution requirement and deadline gets accepted through the admission controller.

2) Resource Reclamation: Now consider that $\bar{E}_i$ is less than E_i. Using ApproximateAC directly, it would be difficult to modify our data structure to reclaim the reserved execution for j_i. (We would have to shift points backwards). Thus, to accommodate early completion, we do a lazy update to the linked list L. We maintain another list Q called the *active jobs list*. This list stores all the jobs whose deadline have not yet elapsed. Let the sum of demands of the jobs in the active job list is called *active demand*. We store the current value of active demand for Q in a variable q.

When a job j_i arrives to the system, the admission controller decides whether to admit the job based on an "inflated" execution time which is the sum of active demand (q) and E_i. If admitted, a node is added to the list Q at the end (in deadline order), and active demand q is increased by E_i; however, we defer the updating of nodes of list L. Upon completion of a job j_i in Q, we reclaim execution in the active demand by increasing q by $E_i - \bar{E}_i$. When the deadline of any job j_i in the active job list elapses, it is removed from the front of Q and the active demand q is reduced by $\bar{E}_i$. Then, a new interval corresponding to $[A_i, d_i]$ is inserted L, with the job's actual execution time $\bar{E}_i$ ($\leq E_i$), and update to existing intervals is performed with $(\delta_x, \delta_y) = (d_i - d_{\text{last}}, \bar{E}_i)$. In this way, it is ensured that if actual execution is less than the estimated, the remaining resource is reclaimed. Also, note that the size of Q is at most the number of active jobs N in the system.

As a final point, many admission controllers have the ability to "reset" upon a system idle point. Clearly, it is desirable to be able to reset the demand of a subsystem to zero at such a point. However, it is not possible to implement such a subsystem reset for demand-curve interface model, where complete global knowledge of the state of all other subsystems comprising the system is unknown to a subsystem; doing so could result in a violation of the interface. Consider a subsystem $\mathcal{S}$ with interface $\mathsf{dbi}(\Lambda, t) = .9t$ and jobs $j_1(0, .9, 1)$ and $j_2(.91, .9, 1)$. If $\mathcal{S}$ executes all of j_1 immediately, then it will go idle at time $.9$. If $\mathcal{S}$ resets at this time, j_2 will be admitted even though j_1 and j_2 together clearly violate $\mathsf{dbi}(\Lambda, 1.91)$.

VII. Simulation

In this section, we evaluate the performance of ExactAC and ApproximateAC by running them over synthetically generated MAD jobs. During simulation we used following parameters and value ranges.

- We use a periodic demand interface with set of 8 periodic tasks with period p and execution e as shown in Table I, and a total utilization of 0.5. The periods are randomly generated in the range $[5, 40]$ and task utilizations are generated using UUniFast algorithm [25]. The generated tasks have a hyperperiod H equal to 197505.
- For MAD job j_i, we generate following parameters from uniform distribution: interarrival time x between succes-

TABLE I
PERIODIC dbi PARAMETERS

p	7	15	9	19	21	27	35	11
e	0.1	1.1	0.18	0.4	0.22	5.2	4	1.7

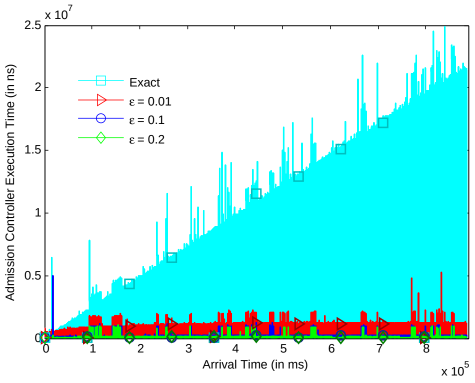


Fig. 11. Execution Time vs Arrival Time of Jobs

sive jobs is in the range $[0, 20]$ (i.e.,$A_i = A_{i-1} + x$); the relative deadline parameter y is in the range $[0, 50]$ (i.e., $D_i = \max\{A_{i-1}, d_{i-1}\} - A_i + y$); and the execution time E_i is in the range $[1, D_i]$.

- A 2.33 GHz Intel Core 2 Duo E6550 machine with 2.0GB RAM is used. The simulation runs until $A_i \geq 4H$.
- We use $\epsilon = [0.01, 0.1, 0.2]$.

We compare our proposed algorithms using two metrics: execution time and the number of accepted jobs over time. Figure 11 shows the execution time trace over time for each of the algorithms. Each point in the plot represents the execution time of corresponding algorithm in nanoseconds for the job arrival at time shown in the horizontal axis. Note, since this plot shows execution time for every run of the algorithm (it might accept or reject the job), the execution time highly fluctuates. Execution time is higher for the "accept" cases than "reject" cases, as it checks every interval in the list and updates all of them. On the other hand, an admission controller might "reject" a job at the very first line (see the algorithms in Section IV, V), or after checking the corresponding list of intervals partially.

The next plot (Figure 12) compares the execution time required for acceptance (i.e., the worst-case) of the algorithms, given that the algorithm has already accepted the number of jobs indicated in the horizontal axis. For example, a point in the red curve at $x = 100$ represents time required by the approximate algorithm ($\epsilon = 0.01$) to admit a job in the system with already 100 jobs in the system. Note that set of accepted jobs for different algorithms will be different, however, the time required by the algorithm does not depend on job specific parameters, rather on the number of already accepted jobs by that algorithm. The plot certifies this by showing linear growth in the execution time for the exact and saturated execution time for the approximate algorithms. Thus, we have a significant reduction in running time for the approximate admission controller over the infeasible exact

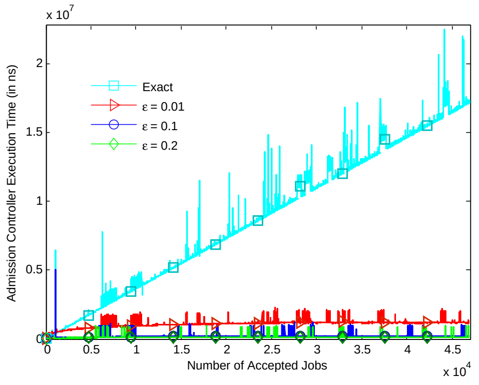


Fig. 12. Execution Time vs Number of Accepted Jobs

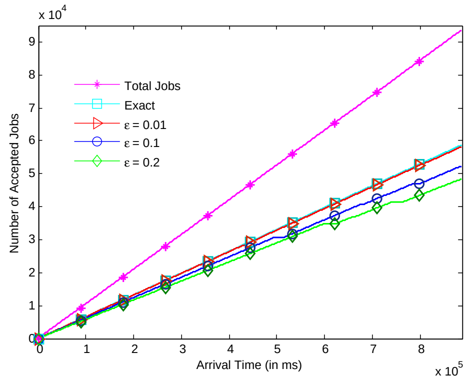


Fig. 13. Number of Accepted Jobs vs Arrival Time of Jobs

admission controller.

The last plot shown in Figure 13 compares the number of jobs admitted by each of the algorithms over time. The leftmost curve represents total number of jobs that arrived in the system over time. The exact algorithm admits more jobs to the system then the approximate algorithms which is intuitive. We observe that the approximate algorithm with $\epsilon = 0.01$ performs very close to the exact algorithm.

VIII. Admission Controller for Arbitrary Aperiodic Jobs

In this section we relax the constraint of the MAD property for the aperiodic jobs in the system; that is, jobs may arrive in the system at any order of deadline. In this case, when a new job j_k is admitted to the system the demand over all intervals corresponding to the nodes in the list L (xy-plane data structure) will not increase the same amount (i.e., $\delta_y = E_k$). The intervals with earlier deadline than d_k will be incremented and the intervals with later deadline will remain the same. For each new job arrival, the admission controller must ensure that the demand of all preceding and succeeding jobs (in absolute deadline order) considering the execution requirement of the new job is less or equal the dbi.

First, we show via an example why the approach for MAD-sequenced job will not work for arbitrary aperiodic jobs. Consider our two-step dbi example (Section IV, Figure 3) and a sequence of aperiodic jobs $j_1(0, 1, 2), j_2(1, 1, 1.9)$, and

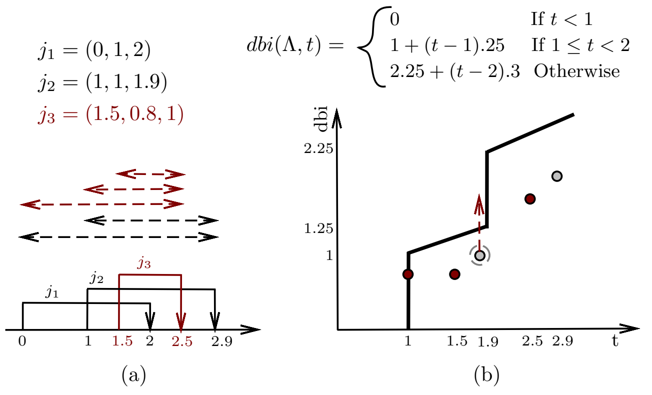

$j_1 = (0,1,2)$
$j_2 = (1,1,1.9)$
$j_3 = (1.5,0.8,1)$
$dbi(\Lambda,t) = \begin{cases} 0 & \text{If } t<1 \\ 1+(t-1).25 & \text{If } 1 \le t < 2 \\ 2.25+(t-2).3 & \text{Otherwise} \end{cases}$
j_1
j_2
j_3
0
1
1.5
2
2.5
2.9
(a)
dbi
2.25
1.25
1
1
1.5
1.9
2.5
2.9
t
(b)

Fig. 14. EXACTAC is insufficient for arbitrary aperiodic jobs.

```
APPROXIMATEAC-A($\Lambda, j_k, \epsilon$)
if dbi($\Lambda, D_k$) $< E_k$
        Reject.
$\delta_x \leftarrow d_k - d_{\text{last}}$; $\delta_y \leftarrow E_k$
Insert a node in $B$ corresponding to $j_k$ in absolute deadline order.
Copy the list $L$ to a list $L'$.
for each node in $B$ in non-decreasing order of $d_i$
        INSERT-A the node in $L'$
        UPDATE-A list $L'$
        if any SHIFT-A or MERGE-A violates dbi
                Delete the node from $B$, delete list $L'$. Reject.
end for
Delete list $L'$. Accept $j_k$.
```

```
UPONJOBDEADLINE-A($j_k$)
INSERT-A($D_k - \delta_x, E_k - \delta_y$) in $L$
UPDATE-A($\hat{P}_h, \delta_x, \delta_y, A_k$) list $L$
$d_{\text{last}} \leftarrow d_k$
Delete the node corresponding to $j_k$ from $B$.
```

Fig. 15. Pseudo-code for approximate admission control of arbitrary aperiodic jobs for an arbitrary demand-curve.

```
UPDATE-A($\hat{P}_h, \delta_x, \delta_y, A_k$)
$\hat{P}_p \leftarrow \hat{P}_h$
if $\hat{P}_p.a \le A_k$
        SHIFT-A($\hat{P}_p, \delta_x, \delta_y$)
    else
        SHIFT-A($\hat{P}_p, \delta_x, 0$)
$\hat{P}_c \leftarrow \hat{P}_p$.next
while $\hat{P}_c$ not null
        if $\hat{P}_c.a \le A_k$
                SHIFT-A($\hat{P}_c, \delta_x, \delta_y$)
            else
                SHIFT-A($\hat{P}_c, \delta_x, 0$)
        $j \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_p.y_l) \rceil$; $k \leftarrow \lceil \log_{1+\epsilon}(\hat{P}_c.y_r) \rceil$
        if $j == k$
                MERGE-A($\hat{P}_p, \hat{P}_c$)
        $\hat{P}_p \leftarrow \hat{P}_c$; $\hat{P}_c \leftarrow \hat{P}_c$.next
end while
```

```
MERGE-A($\hat{P}_1, \hat{P}_2$)
if $\hat{P}_1.x_l > \hat{P}_2.x_l$
        $\hat{P}_1.x_l = \hat{P}_2.x_l$; $\hat{P}_1.y_l = \hat{P}_2.y_l$
if $\hat{P}_1.y_r < \hat{P}_2.y_r$
        $\hat{P}_1.x_r = \hat{P}_2.x_r$; $\hat{P}_1.y_r = \hat{P}_2.y_r$
$\hat{P}_1.a = \min\{\hat{P}_1.a, \hat{P}_2.a\}$
$\hat{P}_1$.next = $\hat{P}_2$.next; DELETE-A($\hat{P}_2$)
```

Fig. 16. Pseudo-code for approximate admission control of arbitrary aperiodic jobs for an arbitrary demand-curve.

$j_3(1.5, 0.8, 1)$ shown in Figure 14. For the $[A_i, d_i]$ interval for each j_i, $\mathsf{dbi}(\Lambda, d_i - A_i) > E_i$. The EXACTAC will admit j_1. When j_2 arrives, the algorithm checks if the demand over the intervals $[A_i, D_2], i = 1, 2$ is met. When j_3 arrives (with $d_3 < d_2$), the algorithm will check whether all the intervals ending at d_3 will meet their demand, i.e., the demand over the intervals $[A_i, d_3], i = 1, 2, 3$ will not violate dbi (red points in Figure 14(b)). However, if j_3 is admitted the demand over the interval $[A_2, d_2]$ will also increase by E_3 amount since $A_2 < A_3$ and $d_2 > d_3$, and this will cause a violation of the dbi, which EXACTAC will not be able to determine. Therefore, we need a procedure to check the demand over the intervals ending after the new job's deadline.

For admission control purposes, a data structure containing the admitted active jobs ordered by their absolute deadline needs to be maintained (similar to [20]). A straightforward extension of our APPROXIMATEAC for MAD aperiodic jobs to approximate admission control of arbitrary aperiodic jobs is given in APPROXIMATEAC-A (Figure 15). We describe update and merge operations in UPDATE-A and MERGE-A respectively. Other operations (e.g., initialize, insert, shift, delete) are similar to the corresponding subroutines shown in Figure 7 (we denote all the subroutines with a suffix '-A' in this section). The UPONJOBDEADLINE-A represents the procedure invoked when the deadline of any node in B elapses.

We give a sketch of the steps for admission control of arbitrary aperiodic jobs for an arbitrary demand-curve interface.

- For the nodes in the list L, we store one more variable a, which corresponds to the arrival time of the interval represented by node $\hat{P}$. When we INSERT-A a new node corresponding to an accepted job j_k in L, we set $a = A_k$. When we MERGE-A($\hat{P}_1, \hat{P}_2$) two nodes, a is set equal to $\min\{\hat{P}_1.a, \hat{P}_2.a\}$, and for SHIFT-A operation we keep it unchanged.
- We keep a data structure B to store all active jobs in the system in their absolute deadline order where each node represents an active job, and stores its arrival A_i, absolute deadline d_i and execution E_i.
- When a new job j_k arrives in the system, we first check if the condition $\mathsf{dbi}(\Lambda, D_k) \ge E_k$ holds. Then, we determine its position in B (in absolute deadline order), and insert it to B if it does not cause a $\mathsf{dbi}(\Lambda, \cdot)$ violation (see last bullet point).

- When the deadline of a job j_k in B elapses, we remove it from the front of B and INSERT-A a node $\hat{P}$ in L corresponding to the interval $[A_k, d_k]$ in non-decreasing order of $\hat{P}.x_l$ of L (UPONJOBDEADLINE-A).
- Then we update the nodes in L by $(\delta_x, \delta_y) = (d_{\text{last}} - d_k, E_k)$ amount which have $\hat{P}.a \leq A_k$, and $(\delta_x, \delta_y) = (d_{\text{last}} - d_k, 0)$ amount which have $\hat{P}.a > A_k$, where d_{last} represents the absolute deadline of the last inserted job (node) in L.
- Although the demand over the newly-inserted job's interval and the SHIFT-A operations do not violate the dbi, a MERGE-A operation can potentially violate the dbi. To overcome this issue, we modify the admission controller as follows. We keep a temporary list L' which is basically a replica of the list L after each call to APPROXIMATEAC-A and UPDATE-A. After job j_k is inserted to B in its absolute deadline order, we simulate the MERGE-A operation on temporary list L' to determine whether the new interval corresponding to j_k, and all the updated intervals in L' as a consequence of inserting j_k to L and shifting the existing intervals will still meet equation 2. For each node in B, we INSERT-A a node in L' in non-decreasing order of $\hat{P}.x_l$ value. Then for each insert, we update L' by shifting the nodes after the newly-inserted node towards right $\delta_x = d_i - d_{last}$ amount, and upwards $\delta_y = E_i$ amount only when $\hat{P}.a \leq A_i$ (UPDATE-A). As we perform updates, we check if two nodes are in the same $(1+\epsilon)$ region, in which case, we MERGE-A the nodes. If any MERGE-A operation causes the dbi to violate, we stop, and reject j_k. Then we delete the temporary list L'.

Essentially, the above approach uses B as a "staging" data structure to force nodes to be added to L in order of absolute deadline. To check whether we may admit a new job, we are essentially simulating the insert operation of the arrived active jobs in deadline order and checking to see if any interface violations occur. Thus, the above extension will add an additional multiplicative factor of N (i.e., the largest length of list B) to the overall time complexity. Future research is needed to determine whether this time complexity can be reduced further with a more sophisticated technique or data structure. Furthermore, we are currently working on a formal proof of correctness of our non-MAD approach.

IX. CONCLUSION

In this paper we addressed the problem of enforcing and policing the demand-curve interface for a subsystem of a compositional real-time system. Given a complex, arbitrary demand interface, we proposed an exact admission control algorithm to police the subsystem load according to the interface, and showed that for long-running online systems, this approach is infeasible. As an alternative we developed efficient polynomial time approximation algorithm for admission control. Further, to ensure temporal isolation within the subsystem, we proposed a lightweight server to enforce the interface and reclaim unused execution. The development of these techniques should make it possible to utilize the rich theory developed for demand-curve interfaces such as RTC and also ensure strong temporal isolation.

In our proposed algorithms, we have used list data structures to store intervals for the ease of presentation; the complexity of the approximate admission controller can be improved further by using advanced data structures (e.g. AVL tree). As future work, we will use these techniques to reduce the complexity of the approximate admission controller for arbitrary job arrivals. Further, we will develop interface-policing policies for distributed and multiprocessor real-time systems.

REFERENCES

[1] Z. Deng and J. Liu, "Scheduling real-time applications in an Open environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, CA.

[2] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*. IEEE, May 2001, pp. 75–84.

[3] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards hierarchical scheduling in AUTOSAR," in *Proceedings of the 14th IEEE International Conference on Emerging Technologies & Factory Automation*. IEEE Computer Society, 2009, pp. 1181–1188.

[4] B. Andersson, "A preliminary idea for an 8-competitive, $\log_2 \text{dmax} + \log_2 \log_2 1/u$ asymptotic-space, interface generation algorithm for two-level hierarchical scheduling of constrained-deadline sporadic tasks on a uniprocessor," *SIGBED Review*, vol. 8, pp. 22–29, March 2011.

[5] S. Chakraborty, S. Knzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proceedings of the 6th Design, Automation and Test in Europe (DATE)*, March 2003, pp. 190–195.

[6] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2000, pp. 101–104.

[7] S. Baruah, "Component-based design of hard-real-time systems on multiprocessor platforms: Issues and ideas," in *Proceedings of the Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*. Barcelona, Spain: IEEE Computer Society, December 2008.

[8] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems: The International Journal of Time-Critical Computing*, vol. 17, no. 1, pp. 5–22, July 1999.

[9] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, April 2008.

[10] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in *Proceedings of the IEEE Real-time Systems Symposium*. Tucson, Arizona: IEEE Computer Society, December 2007.

[11] E. Wandeler and L. Thiele, "Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling," in *Proceedings of the 5th ACM international Conference on Embedded Software*. New York, NY, USA: ACM, 2005, pp. 80–89.

[12] ——, "Interface-based design of real-time systems with hierarchical scheduling," in *IEEE Real-Time Technology and Applications Symposium*, 2006, pp. 243–252.

[13] L. Thiele, E. Wandeler, and N. Stoimenov, "Real-time interfaces for composing real-time systems," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. New York, NY, USA: ACM, 2006, pp. 34–43.

[14] L. d. Alfaro and T. A. Henzinger, "Interface theories for component-based design," in *Proceedings of the International Workshop on Embedded Software*. London, UK: Springer-Verlag, 2001, pp. 148–165.

[15] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the Real-Time Systems Symposium*. Madrid, Spain: IEEE Computer Society, December 1998, pp. 3–13.

[16] T. seng Tia, J. W.-S. Liu, J. Sun, L. Jun, and R. Ha, "A linear-time optimal acceptance test for scheduling of hard real-time tasks," 1994.

[17] G. Lipari and G. Buttazzo, “Scheduling real-time multi-task applications in an open system,” in *Proceedings of the EuroMicro Conference on Real-Time Systems*. York, UK: IEEE Computer Society, June 1999.

[18] G. Lipari and S. Baruah, “Efficient scheduling of real-time multi-task applications in dynamic systems,” in *Proceedings of the Real-Time Technology and Applications Symposium*. Washington, DC: IEEE Computer Society Press, May–June 2000, pp. 166–175.

[19] B. Andersson and C. Ekelin, “Exact admission-control for integrated aperiodic and periodic tasks.” *Journal of Computer System and Sciences*, vol. 73, no. 2, pp. 225–241, October 2007.

[20] F. Dewan and N. Fisher, “Admission control for real-time demand-curve interfaces,” in *Proceedings of the Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*. Vienna, Austria: IEEE Computer Society Press, November 2011.

[21] P. Kumar, J.-J. Chen, and L. Thiele, “Demand bound server: Generalized resource reservation for hard real-time systems,” in *International Conference on Embedded Software (EMSOFT)*. Taipei, Taiwan: ACM, October 2011.

[22] F. Dewan and N. Fisher, “Efficient admission control for enforcing arbitrary real-time demand-curve interfaces (extended version),” Wayne State University, Technical Report, 2012. [Online]. Available: http://www.cs.wayne.edu/~fishern/papers/RTSS-2012-TR.pdf

[23] A. K. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment,” Ph.D. dissertation, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983, available as Technical Report No. MIT/LCS/TR-297.

[24] S. Baruah, R. Howell, and L. Rosier, “Feasibility problems for recurring tasks on one processor,” *Theoretical Computer Science*, vol. 118, no. 1, pp. 3–20, 1993.

[25] E. Bini and G. Buttazzo, “Biasing effects in schedulability measures,” in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 2004, pp. 196–203.