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

Farhana Dewan  Nathan Fisher
Department of Computer Science, Wayne State University

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 then design an approximation 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.

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 efficient resource usage.

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 this approach is the potential 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, which can be applied for online load reduction, QoS adaption etc. 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 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 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. 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 in this paper 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 implement each of our proposed admission controllers and show that the approximate algorithm is both efficient and precise in comparison to the exact (Section VI).
- We briefly discuss extensions of our approximate admission controller for enforcing temporal isolation between admitted jobs (Section VII). Due to space constraints, this and other related issues are described more thoroughly in an extended version of this paper [9].

We first present the assumed real-time job and interface
models (Section II) and discuss related work (Section III).

II. MODELS, NOTATIONS, & PROBLEM DEFINITION

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 $d_i = A_i + D_i$. 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 $1 \leq i < |J|: A_i \leq A_{i+1}$).

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 $t_i$ if $t_i \in [A_i, A_i + D_i)$. Let $N$ be the maximum number of active jobs in the subsystem at any given time.

As a starting point in our development of an admission controller, in this paper we restrict ourselves to monotonic absolute deadline (MAD) [8] job arrivals. 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, $A_i \leq A_k \iff d_i \leq d_k$. 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 the extended version of this paper [9].

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 $t_1, t_2 \in \mathbb{R} : 0 \leq t_1 < t_2$, the function 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 $[t_1, t_2]$.

$$\text{demand}(J, t_1, t_2) = \sum_{j \in J} (A_i \geq t_1) \wedge (d_i \leq t_2) E_i.$$ (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 $\Lambda$. Numerous real-time interface models have been proposed in recent years (e.g., the periodic resource model [10], real-time calculus [6], etc.); however, in this paper, we consider $\Lambda$ 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 $\Lambda$. The challenge of generating and composing demand-curve interfaces [6] is important, but orthogonal to the problem addressed in this paper.

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 $\Lambda$ 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 dbi$(\Lambda, t)$. Formally, the dbi ensures that the following condition holds

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

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

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 [10], [11] can be used as admission controllers, we do not assume any specific scheduling algorithm in this paper. We only require that the underlying scheduling algorithm meets all deadlines if the admitted set of jobs satisfy the specified demand interface.

C. Problem Statement

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

Given an interface $\Lambda$ 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 [12], [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 [12], [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, but fundamentally differs from our 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. 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 ordering of 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 and prove its correctness.

A. Algorithm Description

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 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 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 $d_{last}$.

When a new job $j_k$ arrives to the system, the demand of the intervals $[A_i, d_k]$, $\forall j_i \in J$ 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 $d_k$ are obtained by incrementing $x$ values in $S$ by $\delta_x = d_k - d_{last}$, and the demand over these intervals can be obtained by incrementing $y$ values in the set by $\delta_y = E_k$ amount. Figure 2 visualizes the scenario after job $j_4$ arrives to the system with already accepted jobs $j_1, j_2$ and $j_3$. For all the new intervals with the increment, the algorithm checks if the demand in the interval is less than or equal the dbi (Line 3 to Line 7). If for any interval this condition is violated, $j_k$ is rejected and the point corresponding to $j_k$ 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 incrementing $(\delta_x, \delta_y)$ amount in the respective dimensions (Line 9 to Line 12).

Clearly, exact admission control for arbitrary demand-curves is computationally linear in the number of jobs that have arrived in the system. Since all past intervals need to be verified, the number of such intervals becomes 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 approach is not applicable to arbitrary
demand curves. We will address this major drawback by designing an approximate admission controller in Section V.

B. Proof of Correctness

We now provide the main theorem which states 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($\Lambda, j_k$) returns “Accept”, if and only if, $j_k$ may be admitted without $J \cup \{j_k\}$ violating $\Lambda$.

We need to prove the next three lemmas in order to prove the theorem. The first lemma shows that we only need to check a finite number of intervals (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,

$$\forall j_i, j_k \in J : d_i \leq d_k \implies \text{demand}(J, A_i, d_{i\ast}) \leq \text{dbi}(\Lambda, d_k - A_i).$$

Proof: The “only if” direction of the lemma is trivial as $A_i$ and $d_k$ are elements of $\mathbb{R}$ and $A_i \leq d_k$. 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,

$$\text{demand}(J, t_1, t_2) \leq \text{dbi}(\Lambda, t_2 - t_1).$$

(3)

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 $\infty$. Consider two partitions of the interval $[0, \infty)$ into two sets of subintervals $(A_i, A_i]$ where $1 \leq i \leq |J| + 1$ and $[d_i, d_{i\ast}]$ where $0 \leq \ell \leq |J|$. Since $0 \leq t_1$, there exists some $i : (1 \leq i \leq |J| + 1)$ where $t_1 \in (A_{i\ast-1}, A_i]$. Observe that demand($J, t_1, t_2$) = demand($J, A_i, t_2$) since no jobs arrive in the interval $(A_{i\ast-1}, A_i)$. Similarly, there exists some $i : (0 \leq \ell \leq |J|)$ where $t_2 \in [d_\ell, d_{\ell\ast}]$ and demand($J, A_i, t_2$) = demand($J, A_i, d_\ell$). Thus, demand($J, t_1, t_2$) = demand($J, A_i, d_\ell$). By Equation 3, demand($J, t_1, t_2$) $\leq$ dbi($\Lambda, d_{\ell\ast} - A_i$) is true. Since $d_{\ell\ast} \leq t_1 \leq A_i$, and dbi is monotonically non-decreasing, it must be that dbi($\Lambda, d_{\ell\ast} - A_i$) $\leq$ dbi($\Lambda, t_2 - t_1$). These last two statements together imply demand($J, t_1, t_2$) $\leq$ dbi($\Lambda, t_2 - t_1$), contradicting Equation 4. Thus, the lemma is true. \[Box\]

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., $d_{\text{last}}$).

Lemma 2: After the call to EXACTAC-INIT() and the $k$’th invocation of EXACTAC($\Lambda, j_k$) where $k \in \mathbb{N}$, for MAD-sequence jobs $j_1, j_2, \ldots, j_k$, define $J_k$ to be the set of jobs admitted by the algorithm. For each job $j_i \in J_k$, $\exists (x, y) | x = d_{\text{last}} - A_i, y = \text{demand}(J_k, A_i, d_{\text{last}})$. Furthermore, $d_{\text{last}}$ equals max{$\emptyset$} $\cup \{d_i | j_i \in J_k\}$.

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 $d_{\text{last}}$ is initialized to zero.

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

Inductive Step: We must show that the lemma holds for the $k$’th call to EXACTAC($\Lambda, j_k$). The admission controller can either return “accept” or “reject”. We first consider the case when EXACTAC($\Lambda, j_k$) returns “reject”. Then, $J_{k-1}$ is identical to $J_k$ and $d_{\text{last}}$ 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 the call to EXACTAC($\Lambda, j_{k-1}$).

Now, consider the case when EXACTAC($\Lambda, j_k$) returns “accept”. Line 13 of the procedure sets $d_{\text{last}}$ equal to $d_{k\ast}$; Let the updated value of $d_{\text{last}}$ and $S$ be denoted by $d_{\text{new}}$ and $S_{\text{new}}$, respectively. Let $d_{\text{old}}$ and $S_{\text{old}}$ denote the value of the $d_{\text{last}}$ and $S$ variables, prior to EXACTAC($\Lambda, j_k$). By the inductive hypothesis, each job $j_k \in J_{k-1}$, there exists $(x, y) \in S_{\text{old}}$ such that $x$ equals $d_{\text{old}}$ $- A_i$ and $y$ equals $\text{demand}(J_{k-1}, A_i, d_{\text{old}})$. The for-loop of Lines 9 to 12 shifts each point $(x, y)$ to the right by $\delta_x = d_{\text{new}} - d_{\text{last}}$ and up by $\delta_y = E_k$. Thus, each $(x, y) \in S_{\text{old}}$ that corresponds to $j_k \in J_{k-1}$ is now $(x + \delta_x, y + \delta_y) \in S_{\text{new}}$. Furthermore, $x + \delta_x$ equals $(d_{\text{new}} - d_{\text{last}}) + d_{\text{last}} - A_i = d_{\text{new}} - A_i$ and $y + \delta_y$ equals $\text{demand}(J_{k-1}, A_i, d_{\text{new}}) + E_k$. The last expression is equivalent to $\text{demand}(J_k, A_i, d_{\text{new}})$ since increasing the interval length by $\delta_x$ includes only the new job $j_k$ in the interval $[A_i, d_{\text{new}}]$. Finally, adding the point $(d_k - \delta_x, E_k - \delta_y)$ in Line 2 and shifting by $\delta_x$ and $\delta_y$ is equivalent to adding $(d_k, E_k)$ which equals $(d_{\text{new}} - A_k, \text{demand}(J_k, A_k, d_{\text{new}}))$. 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 $d_{\text{last}}$ ($\leq d_k$) in $S$, we may update the points in $S$ by shifting them to the right the $\delta_x = d_k - d_{\text{last}}$ 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 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. As the lemma’s proof is similar in structure to Lemma 2, we omit it for space (see [9]).

Lemma 3: After the call to EXACTAC-INIT() and the $k$’th invocation of EXACTAC($\Lambda, j_k$) where $k \in \mathbb{N}$, for MAD-sequence 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 $\Lambda$. Furthermore, if EXACTAC($\Lambda, j_k$) returns “reject”, then $j_k \notin J_k$, then Equation 2 is false for $J_{k-1} \cup \{j_k\}$.

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

V. APPROXIMATE ADMISSION CONTROL FOR MAD JOBS

As mentioned in the previous section, the exact admission controller is intractable for long-running systems. Unfortu-
nately, it is easy to show that naively eliminating demand pairs in $S$ may result the admission controller to incorrectly admit a job to the system. (See extended paper [9] for an example). 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 points (intervals) stored by the algorithm using a more sophisticated approach than just naively dropping intervals. We achieve our reduction in time complexity for admission control via four main steps:

1) Divide the $xy$-Plane into Regions: We divide the $xy$-plane into increasingly large intervals based on a user-supplied accuracy parameter $\epsilon > 0$. A smaller value of $\epsilon$ 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 $A^i$ 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 $A^i$. Formally, the $i$'th $(1 + \epsilon)$-region is defined as

$$A^i \overset{\text{def}}{=} \{(x, y) \in \mathbb{R}^2 \mid (1 + \epsilon)^{i-1} \leq y < (1 + \epsilon)^i\} \quad (5)$$

Figure 3 gives a visual depiction of the horizontal division of the $xy$-plane. We denote the lower bound and upper bound of region $A^i$ as $A^i.lb$ and $A^i.ub$ respectively.

![Figure 3. Approximating Y-axes of dbi.](image)

2) Merge Intervals Within A Region: Consider two distinct intervals represented by $(x_1, y_1)$ and $(x_r, y_r)$ in the same $(1 + \epsilon)$-region $A^i$. To reduce the number of intervals stored, we merge two such intervals into an approximation point.

Definition 4 (Approximation Point $\hat{P}$): Consider two points $(x_1, y_1)$ and $(x_r, y_r)$ where $x_1 \leq x_r$ and $y_1 \leq y_r$. We define the approximation point $\hat{P}((x_1, y_1), (x_r, y_r))$ “anchored” by points $(x_1, y_1)$ and $(x_r, y_r)$ as

$$\hat{P}((x_1, y_1), (x_r, y_r)) \overset{\text{def}}{=} (x_1, y_r). \quad (6)$$

The points $(x_1, y_1)$ and $(x_r, y_r)$ are referred to as the left-anchor and right-anchor of approximation point $\hat{P}$.

For simplicity, we drop the “$((x_1, y_1), (x_r, y_r))$” from $\hat{P}$ when it is clear which approximation point we are referring to. The notation $\hat{P}.x_1$ and $\hat{P}.y_r$ (respectively, $\hat{P}.x_r$ and $\hat{P}.y_r$) refer to the $x$ and $y$ coordinates of the left (right) anchor point. Figure 4(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 $\hat{P}.y_r \leq (1 + \epsilon)\hat{P}.y_r$. In other words, the $y$-value of an 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.

3) Eliminate Redundant Points: Observe from Figure 4(a) that the region below and to the right of the approximation point $\hat{P}$ forms an infinitely-extending rectangular region.

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

$$R(\hat{P}) \overset{\text{def}}{=} \{(x, y) \in \mathbb{R}^2 \mid (\hat{P}.x_l \leq x) \land (\hat{P}.y_r \geq y)\}. \quad (7)$$

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 $\text{dbi}(\Lambda, \hat{P}.x_l) - \hat{P}.y_r \leq \text{dbi}(\Lambda, x) - y$.

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

Therefore, we can conclude that if approximate point $\hat{P}$ is below dbi, then any point in the redundancy region $R(\hat{P})$ 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 $A^i$ may eventually move (completely with both anchor points) into another region $A^j$ where $j > i$. We say that an approximation point $\hat{P}$ is “completely in” $A^i$ if $A^i.lb \leq \hat{P}.x_l \leq \hat{P}.y_r \leq A^i.ub$; otherwise, the approximation point “straddles” $(1 + \epsilon)$-regions. An approximation point $\hat{P}$ is “contained” (not necessarily completely) in $A^i$, if $A^i.lb \leq \hat{P}.y_r \leq A^i.ub$ Consider any two approximation points $\hat{P}_1$ and $\hat{P}_2$ that are completely in $A^i$: i.e., for $k \in \{1, 2\}$, the approximation points $\hat{P}_k$ satisfies $\hat{P}_k.y_r \geq A^i.lb$ and $\hat{P}_k.y_r \leq A^i.ub$. 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 4(b) depicts the merge of two approximation points and evolution of their redundancy regions after merging.
of nodes, where \( \hat{\} \) represents an approximation point. Each node refers to both a point \( xy \) represents the head of the list. At any time the algorithm to if no such node exists. The nodes of in the list or \( \hat{\} \) to zero and \( L \) to the next node in the list. We (\( \hat{\} \) has \( Y \) \( D \) \( \hat{\} \) in \( E \) is the shift in \( X \)-axes and \( \delta_y \) is the shift in \( Y \)-axes. The \textsc{Merge}(\( P_1 \), \( P_2 \)) operation merges two nodes by updating the left and right anchor of \( P_1 \) with the left-most of the two left anchors, and the top-most of the two right anchors respectively (Figure 4(b)). Then it deletes the node \( P_2 \) from the list \( L \). When a new job \( j_k \) (\( A_k, E_k, D_k \)) arrives in the system, \textsc{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., \( \text{dbi}(\Lambda, D_k) \). It is possible to show (similar to Lemma 2) that all the intervals of interest for \( \text{MAD} \) 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 \( \delta_x = d_k - d_{last} \) and the increase in demand is \( \delta_y = E_k \). In the \text{while}-loop of Lines 7 to 15, we check that each of the approximate points will still fall below the \text{dbi}(\Lambda, \cdot) if they are shifted to the right by \( \delta_x \) and upwards by \( \delta_y \). In particular, Line 8 determines if the individual points shifted will violate \( \Lambda \); 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 \( \Lambda \). 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 \( L \), the algorithm “accepts” the job and performs \textsc{Update}(\( P_h \), \( \delta_x \), \( \delta_y \)) operation for all nodes starting from the head of the list. This procedure uses \textsc{Shift}(\( \delta_x \), \( \delta_y \)) operation to shift each node (both left and right anchors), and performs \textsc{Merge} operation on two consecutive nodes in the same \((1 + \epsilon)\)-region.

### 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 \textsc{ApproximateAC}(\( \Lambda, j_k, \epsilon \)) has returned from its execution.

**Theorem 2:** If \textsc{ApproximateAC}(\( \Lambda, j_k, \epsilon \)) returns “Accept”, then \( j_k \) may be admitted without violating \( \Lambda \).

Our first lemma of this subsection shows that any deleted point is “covered” by another point; i.e., the deleted point must be contained in the redundancy region of a node in the list \( L \).

**Lemma 5:** For any approximation point \( \hat{\} \) that was inserted into list \( L \), if \( \hat{\} \) is deleted, then there exists some approximation point \( \hat{\} \) in \( L \) such that \( \hat{\} \leq \hat{\} \) and \( \hat{\} \geq \hat{\} \).

**Proof:** A node is deleted only in Line 9 or 13 of \textsc{ApproximateAC}(\( \Lambda, j_k, \epsilon \)) or in the \textsc{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 \textsc{Merge} subroutine. Recall that when we merge approximation points \( P \) and \( P \) in 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 \textsc{Delete}(\( \hat{\} \)) operation deletes node \( \hat{\} \) from the list. A \textsc{Shift}(\( \hat{\} \), \( \delta_x \), \( \delta_y \)) operation shifts both the left anchor and the right anchor of \( \hat{\} \) by \( \delta_x \) amount, where \( \delta_x \) is the shift in \( X \)-axes and \( \delta_y \) is the shift in \( Y \)-axes. The \textsc{Merge}(\( P_1 \), \( P_2 \)) operation merges two nodes by updating the left and right anchor of \( P_1 \) with the left-most of the two left anchors, and the top-most of the two right anchors respectively (Figure 4(b)). Then it deletes the node \( P_2 \) from the list \( L \). When a new job \( j_k \) (\( A_k, E_k, D_k \)) arrives in the system, \textsc{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., \( \text{dbi}(\Lambda, D_k) \). It is possible to show (similar to Lemma 2) that all the intervals of interest for \( \text{MAD} \) 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 \( \delta_x = d_k - d_{last} \) and the increase in demand is \( \delta_y = E_k \). In the \text{while}-loop of Lines 7 to 15, we check that each of the approximate points will still fall below the \text{dbi}(\Lambda, \cdot) if they are shifted to the right by \( \delta_x \) and upwards by \( \delta_y \). In particular, Line 8 determines if the individual points shifted will violate \( \Lambda \); 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 \( \Lambda \). 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 \( L \), the algorithm “accepts” the job and performs \textsc{Update}(\( P_h \), \( \delta_x \), \( \delta_y \)) operation for all nodes starting from the head of the list. This procedure uses \textsc{Shift}(\( \delta_x \), \( \delta_y \)) operation to shift each node (both left and right anchors), and performs \textsc{Merge} operation on two consecutive nodes in the same \((1 + \epsilon)\)-region.

### A. Algorithm Description

In Figure 5, we present the pseudocode for our approximate admission control algorithm for an arbitrary demand-curve interface. The algorithm keeps a linked list \( L \) of nodes, where each node \( \hat{\} \) represents a linked point. Each node \( \hat{\} \) consists of two points: left anchor \( (x_l, y_l) \) and right anchor \( (x_r, y_r) \), and a pointer next to the next node in the list. We abuse notation somewhat to allow \( \hat{\} \) to refer to both a point in the \( xy \)-plane and a node in \( L \); \( \hat{\} \) next is the next node after \( \hat{\} \) 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. \( \hat{\} \) represents the head of the list. 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. (Pseudocode [9] is not provided as their implementations are straightforward). The procedure \textsc{ApproximateAC-Init()} initializes the list \( L \) to empty, \( d_{last} \) to zero and \( \hat{\} \) to null. An \textsc{Insert}(\( x, y \)) operation creates a node in the list \( L \) with the approximate point equal to
and \( \hat{P}_2 \), a new point \( \hat{P}' \) is created. It is obvious that \( \hat{P}_1 \) and \( \hat{P}_2 \) are in the redundancy region \( R(\hat{P}') \) (refer to Figure 4(b)). Thus, for the two points deleted by the merge operation, by definition of redundancy region, \( \hat{P}'_i \leq \hat{P}_k \) and \( \hat{P}'_i \geq \hat{P}_k \), for \( k \in \{1, 2\} \).

The next two lemmas are equivalent to Lemmas 2 and 3 in Section IV, except that they rely on the observation of Lemma 5. Due to the similarity to these lemmas, we omit their proofs. The complete proofs are in the extended paper [9].

**Lemma 6:** After the call to APPROXIMATE-AC-INIT() and the \( k \)'th invocation of APPROXIMATE-AC(\( \Lambda, j_k, \epsilon \)) 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 job \( j \in J_k \), there exists an approximation point \( \hat{P} \) in list \( L \) such that \( \hat{P} \cdot x_i \) is at most \( d_{last} - A_i \) and \( \hat{P} \cdot y_r \) is at least demand\( (J_k, A_i, d_{last}) \). Furthermore, \( d_{last} \) equals \( \max \{ \{0\} \cup \{ d | j \in J_k \} \} \).

**Lemma 7:** After the call to APPROXIMATE-AC-INIT() and the \( k \)'th invocation of APPROXIMATE-AC(\( \Lambda, j_k, \epsilon \)) 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 \( \Lambda \).

Theorem 2 immediately follows from the above lemma.

**C. Approximation Ratio**

In this section, we argue that when the approximate admission controller rejects a job, then the exact admission controller would also have done so on a slightly “smaller” dbi. The accuracy of the test is determined by the accuracy parameter \( \epsilon > 0 \). Note that, in online setting, set of jobs accepted by the approximate algorithm might not be a subset of set of jobs accepted by the exact algorithm. However, for the ease of analysis, in this section we assume that initially both the algorithms have same set of accepted jobs. 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 bounds the ratio of \( y \)-values of the right and left anchor point in terms of \( \epsilon \).

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

\[
P \cdot y_r \leq (1 + \epsilon) \hat{P} \cdot y_r.
\]

**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, \( \hat{P}' \cdot y_r = \hat{P} \cdot y_r + \delta_y \) and \( \hat{P}' \cdot y_l = \hat{P} \cdot y_l + \delta_y \). If the invariant is true prior to the \( \text{SHIFT} \) call, then \( \hat{P}' \cdot y_r = \hat{P} \cdot y_r + \delta_y \leq (1 + \epsilon) \hat{P} \cdot y_l + \delta_y \leq (1 + \epsilon) \hat{P} \cdot y_r + \delta_y = (1 + \epsilon) \hat{P} \cdot y_r \). Thus, the invariant continues to hold after \( \text{SHIFT} \).

Finally, to see that the invariant holds after MERGE, observe that the algorithm only merges two consecutive points \( \hat{P}_k \) and \( \hat{P}_{k+1} \) in the list \( L \) when both anchor points of these approximation points are completely within some region \( A' \). For consecutive points \( \hat{P}_k \) and \( \hat{P}_{k+1} \) in \( L \), it may be shown (Lemma 11 in Section V-D) that \( \hat{P} \cdot y_r \leq \hat{P}_{k+1} \cdot y_r \). Thisobservation taken together with the fact that the points are completely within \( A' \) implies that \( A' \cdot y_l \leq \hat{P}_k \cdot y_l \leq \hat{P}_{k+1} \cdot y_l \leq A' \cdot y_b \). Thus inequality follows from Definition 3. Since \( A' \cdot y_l \leq \hat{P}_k \cdot y_l \), it must be that \( \hat{P}_{k+1} \cdot y_r \leq (1 + \epsilon) A' \cdot y_l \). The MERGE operation will make a new approximation point \( \hat{P}' \) with \( \hat{P}' \cdot y_r = \hat{P} \cdot y_r \) and \( \hat{P}' \cdot y_l = \hat{P}_{k+1} \cdot y_l \); thus, the invariant remains true.

The following corollary follows from the observation that a left anchor point below \( A^{-1} \cdot y_l \) and a right anchor point above \( A^{-1} \cdot y_b \) (i.e., the approximation point spans three or more regions) would violate the invariant of Lemma 8.

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

The next lemma shows that for any approximation point, the left anchor corresponds to the exact demand over some interval. The proof is nearly identical to Lemma 2.

**Lemma 9:** After the call to APPROXIMATE-AC-INIT() and the \( k \)'th invocation of APPROXIMATE-AC(\( \Lambda, j_k, \epsilon \)) where \( k \in \mathbb{N} \), for MAD-sequenced jobs \( j_{1}, j_{2}, \ldots , j_{k} \), let \( J_k \) be the set of jobs admitted by the approximate admission controller. For each \( \hat{P}_i \) in \( L \), it must be that there exists a \( j \in J_k \) such that \( \text{demand}(J, A_i, d_{last}) \) equals \( \hat{P} \cdot y_l \) and \( d_{last} - A_i \) equals \( \hat{P} \cdot x_i \).

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 APPROXIMATE-AC(\( \Lambda, j_k, \epsilon \)) returns “Reject”, then EXACTAC(\( \Lambda, j_k \)) also returns “Reject” for a demand-curve \( \frac{\text{dibi}(\Lambda, \cdot)}{\text{dibi}(\cdot, \cdot)} \) on the same previously-admitted job set.

**Proof:** If APPROXIMATE-AC(\( \Lambda, j_k, \epsilon \)) 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 \( \text{SHIFT} \) 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, in this case, \( j_k \) would also fail the test of Line 4 of EXACTAC.

By Lemma 9, \( \hat{P} \cdot x_i \) equals \( d_{last} - A_i \) and \( \hat{P} \cdot y_l \) equals demand\( (J, A_i, d_{last}) \) for some \( j \in J \). If \( \hat{P} \) fails in Line 8, then \( \text{dibi}(\Lambda, \hat{P} \cdot x_i + \delta_x) < \hat{P} \cdot y_r + \delta_y \) which implies that \( \text{dibi}(\Lambda, d_{k} - A_i) < \hat{P} \cdot y_r + \delta_y \leq (1 + \epsilon) \hat{P} \cdot y_r + \delta_y \). Since \( \delta_x = (1 + \epsilon) \text{demand}(J, A_i, d_{last}) + E_k \leq (1 + \epsilon) \text{demand}(J \cup \Lambda) \), we have that \( \text{dibi}(\Lambda, \hat{P} \cdot x_i + \delta_x) \).
\[ \{j_k\}, A_r, d_k \] 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. \[ \Box \]

D. Algorithm Complexity

The complexity of the approximate admission controller depends on the size of the linked list \( L \), since both \textsc{ApproximateAC} and \textsc{Update} have a loop traversing the entire list. We argue in Theorem 4 that the number of approximation points (nodes in \( L \)) for each \((1 + \epsilon)\)-region is constant. Therefore, both the time and space complexity of the algorithm is directly proportional to the number of \((1 + \epsilon)\)-regions. The time or space required by the algorithm can be adjusted by tuning the approximation parameter \( \epsilon \).

Theorem 4: For all \( i \), a \((1 + \epsilon)\)-region \( \Lambda_i \) contains at most two approximation points.

Our first lemma shows that each newly-admitted job corresponds to a node at the front of list \( L \). The proof is trivial.

Lemma 10: After the call to \textsc{ApproximateAC-Init()} and the \( k \)'th invocation of \textsc{ApproximateAC}(\( \Lambda, j_k, \epsilon \)) 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 newly-admitted approximation point \( \hat{P} = (D_k, E_k) \) is the first node of list \( L \).

Based on this observation, the following lemma states that 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:

\[ \hat{P}_1.y_r \leq \hat{P}_1.y_r \leq \hat{P}_2.y_l \leq \hat{P}_2.y_r. \]  

(9)

Proof: Observe that the operations that change an approximation point are \textsc{Insert}, \textsc{Shift}, and \textsc{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 \textsc{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 \textsc{Shift} operation for any two approximation points \( \hat{P}_1 \) and \( \hat{P}_2 \). The \textsc{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 \textsc{Merge} operation, consider two successive points \( P \) and \( P' \) 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 \textsc{Merge} (under the assumption that the invariant holds) we had \( P.y_l \leq \hat{P}_1.y_l \leq P'.y_l \leq P'.y_r \leq P_2.y_l \leq P_2.y_r \). After the call to \textsc{Merge}, the approximation point \( \hat{P}_1 \) has left anchor point \( (\hat{P}_1.x_l, \hat{P}_1.y_l) \) and right anchor point \((\hat{P}_1.x_r, \hat{P}_1.y_r)\). Thus, the invariant continues to hold for this case. The lemma may be shown symmetrically if \( P_2 \) precedes \( P \) and \( P' \).

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

Lemma 12: For all \( i \), \((1 + \epsilon)\)-region \( \Lambda_i \) contains at most one approximation point such that \((\Lambda_i.l_b \leq \hat{P}_1.y_r \leq \Lambda_i.u_b) \land (\Lambda_i.l_b \leq \hat{P}_2.y_r \leq \Lambda_i.u_b) \) after each call to \textsc{ApproximateAC}.

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

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 \( \Lambda_i.l_b \) for any \( i > 1 \). This corollary follows from 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 9 of Lemma 11.

Corollary 2: For all \( i \), \( \Lambda_i \) contains at most one approximation point such that \((\Lambda_i.l_b \leq \hat{P}_1.y_r \leq \Lambda_i.u_b) \land (\hat{P}_2.y_l \leq \Lambda_i.l_b) \).

Theorem 4 follows from Lemma 12 and Corollary 2.

Despite bounding the number of approximation points per \((1 + \epsilon)\)-region, the number of these regions could be potentially infinite for an arbitrary \( \text{dbi} \). Typically, there are two design choices: 1) \( \text{dbi} \) is generated from some finite set of recurring tasks and each point can be calculated using some known closed-form equation; or 2) \( \text{dbi} \) is stored as a finite set of linear segments (i.e., the \( \text{dbi} \) has a finite number of steps and an entry for each step in the function) . In [9], we explore the first option by giving a non-trivial subroutine \( \text{dbi-WrapCheck} \) after the \textsc{Update} subroutine in our admission controller to handle a \( \text{dbi} \) generated from periodic/sporadic task systems. For now, we will assume that we are given a finite number of line segments as \( \text{dbi} \) after the \textsc{Shift} of each node.

Let \( r \) be the number of line segments required to specify the \( \text{dbi} \) for \( \Lambda \). We can view the \( \text{dbi} \) as an ordered set \( \{(a_i, b_i), s_i | 1 \leq i \leq r \} \) 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 \( \text{dbi}(\Lambda, t) = b_i + (t - a_i)s_i \). Furthermore, to ensure that \( \text{dbi} \) 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 \( \hat{P}_1.x_l \) larger than \( a_r \), we may modify \textsc{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 $F.x_i \geq \alpha_r$. Otherwise, let $\mathcal{U}$ equal $b_{r-1} + (a_r - \alpha_{r-1})s_{r-1}$. For all points before $a_r$, APPROXIMATEAC requires at most $|\log b_{r+1} \mathcal{U}|$ different $(1+\epsilon)$-regions to cover all the 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 $O(|\log b_{r+1} \mathcal{U}| (|\log r|))$. 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.

VI. SIMULATION

We evaluated the performance of EXACTAC and APPROXIMATEAC by running them over synthetically generated MAD jobs. During simulation the following parameters were used.

- We have used a periodic demand interface obtained from the sum of demand bound functions of 8 periodic tasks with randomly generated periods (in the range $[5,40]$) and execution times with a total utilization of $0.5$. The generated tasks have a hyperperiod $H$ equal to 197505. (See [9] for specific details on each generated task).
- For MAD job $j_i$, we generate following parameters from uniform distribution: inter-arrival time $x$ between successive 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., $D_i = \max\{A_i,d_{i-1}\} - A_i + y$); and the execution time $E_i$ is in the range $[1,D_i]$. We use $\epsilon = [0.01,0.1,0.2]$.
- 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 compare our proposed algorithms using two metrics: execution time and the number of accepted jobs over time. Figure 6 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 (in ms) 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. The execution time is higher for the “accept” cases than the “reject” cases, as it checks every interval in the list and updates all of them.

The time required by the exact and the approximate algorithm does not depend on job specific parameters, rather on the number of already accepted jobs for the exact, and number of approximation regions for the approximate algorithm. The plot certifies this by showing linear growth in execution time for the exact algorithm and execution time with a low saturation point for the approximate algorithms. Thus, we have a significant reduction in running time for the approximate admission controller over the infeasible exact admission controller.

The next plot shown in Figure 7 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.

VII. DISCUSSION

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 $\hat{E}_i$ as the actual execution time of job $j_i$. In this section we address how the overrun ($E_i < \hat{E}_i$) and underrun ($E_i > \hat{E}_i$) situations can be handled so that the interface is not violated.

Each admitted job can be executed within a lightweight server with budget equals the worst-case execution time of the job and deadline equals the relative deadline of the job. The server budget is consumed as the job executes, and when the budget is exhausted (equals zero) the job is suspended. In this way overrun situations can be handled and temporal isolation within subsystem is enforced.

When $\hat{E}_i > E_i$, the remaining unused demand can be reclaimed. Using APPROXIMATEAC directly, it would be difficult to modify our data structure to reclaim the reserved
execution for \( j_k \). (We would have to shift points backwards). Thus, to accommodate early completion, we do not immediately insert a job into linked list \( L \). Instead, we maintain another list \( Q \) called the active jobs list, to store all the jobs whose deadline have not yet elapsed. Let the sum of estimated executions of the jobs in \( Q \) is called active demand. At any time we store the current value of active demand for \( Q \) in a variable \( \lambda \). When a job \( j_k \) arrives to the system, the admission controller decides whether to admit the job based on an “inflated” execution time which equals \( \bar{E}_k \). If admitted, a node is added to the list \( Q \) at the end (in absolute deadline order), and active demand \( q \) is increased by \( E_k \); however, we defer the update of nodes in \( L \). When the deadline of job \( j_k \) in \( Q \) elapses, it is removed from the front of the list and \( q \) is reduced by \( E_k \). Note that the server discussed earlier in this section will ensure that the hard-deadline scheduler does not execute \( j_k \) more than \( E_k \) time units. Then, a new interval corresponding to \([A_k, d_k]\) is inserted to \( L \), with the job’s actual execution time \( E_k \leq E_k \), and \( \text{UPDATE} \) operation is performed with \((\delta_s, \delta_q) = (d_k - d_{last}, E_k)\). In this way, it is ensured that if the actual execution is less than the estimated, the remaining resource is reclaimed. 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 \( S \) with interface \( \text{dbi}(\Lambda, t) = 9t \) and jobs \( j_1(0, 9, 1) \) and \( j_2(91, 9, 1) \). If \( S \) executes all of \( j_1 \) immediately, then it will go idle at time \( 91 \) if \( S \) resets at this time, \( j_2 \) will be admitted even though \( j_1 \) and \( j_2 \) together clearly violate \( \text{dbi}(\Lambda, 1.91) \).

VIII. 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. 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. We are currently working on improving the time/space complexity of our admission controller by using advanced data structure and/or obtaining subsystem reset points to reduce number of stored points. As future work, we will provide interface-policing policies for distributed and multiprocessor real-time systems.

REFERENCES


