CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-Service

Sudipto Das[§] Vivek R. Narasayya[§] Feng Li[†][*] Manoj Syamala[§]

[§]Microsoft Research
Redmond, WA USA
{sudiptod, viveknar, manojsy}@microsoft.com

[†]National University of Singapore
Singapore
li-feng@comp.nus.edu.sg

ABSTRACT

Multi-tenancy and resource sharing are essential to make a Database-as-a-Service (DaaS) cost-effective. However, one major consequence of resource sharing is that the performance of one tenant's workload can be significantly affected by the resource demands of co-located tenants. The lack of performance isolation in a shared environment can make DaaS less attractive to performance-sensitive tenants. Our approach to performance isolation in a DaaS is to isolate the key resources needed by the tenants' workload. In this paper, we focus on the problem of effectively sharing and isolating CPU among co-located tenants in a multi-tenant DaaS. We show that traditional CPU sharing abstractions and algorithms are inadequate to support several key new requirements that arise in DaaS: (a) absolute and fine-grained CPU reservations without static allocation; (b) support elasticity by dynamically adapting to bursty resource demands; and (c) enable the DaaS provider to suitably trade-off revenue with fairness. We implemented these new scheduling algorithms in a commercial DaaS prototype and extensive experiments demonstrate the effectiveness of our techniques.

1. INTRODUCTION

Relational Database-as-a-Service (*DaaS*) providers, such as Microsoft SQL Azure, host large numbers of applications' databases (or *tenants*). Sharing resources among tenants is essential to make DaaS cost-effective. *Multi-tenancy* in the database tier is therefore critical in a DaaS setting. However, one major concern arising from multi-tenancy is the lack of performance isolation. When workloads from multiple tenants contend for shared resources, one tenant's performance may be affected by the workload of another tenant. A natural question to ask is: *what meaningful assurances can a provider of a multi-tenant DaaS platform expose and yet be cost-effective?*

Applications often reason about performance at the level of query latency and/or throughput. One approach is to provide performance assurances in a multi-tenant environment where the provider models application's workload and/or resource requirements to judiciously co-locate tenants such that their performance goals are satisfied [4, 6]. However, if the service provider does not control or restrict the tenant workloads and aims to support general SQL workloads, such workload-level assurances are hard even when the entire server is dedicated to a tenant. Furthermore, tenants' workloads and resource requirements change with time as the workload, access patterns, data sizes, and distributions change. In such a dynamic multi-tenant setting, if a tenant's performance goals are not met, it becomes hard to determine whether the performance degradation resulted from a tenant's change in behavior or due to the provider's fault. Therefore, our approach is to provide *resource isolation* assurances to tenants of a DaaS [15] that isolates a tenant's resources (and hence performance) from resource demands of other concurrently executing tenants. In spite of the large body of existing work on resource management, there are several new requirements in a DaaS which motivate us to rethink the resource management abstractions and mechanisms for isolation.

1.1 Requirements

First, resource assurances should be **absolute**, i.e., independent of other tenants. This exposes to a tenant the paradigm similar to that in a traditional enterprise setting where the tenant provisions a server with adequate resources for their workload. Service differentiation using priorities (common in OS schedulers) or proportions (such as [20, 24]) only provide relative assurances. The *actual* resources allocated to a tenant depends on which other co-located tenant workloads are contending for resources. In a shared environment where different un-related tenants are co-located, such relative assurances are not sufficient for performance-sensitive tenants. Similarly, maximum resource limits, supported by many commercial databases today, does not assure a tenant how much resources will *actually* be allocated under heavy resource contention. A *minimum* resource assurance for each tenant is necessary for performance isolation.

Second, providers want the service to be **cost-effective**. Thus, although the resource assurances are absolute, a strategy of statically reserving resources for a tenant at the granularity of processor cores, disks, memory etc., limits consolidation and increases the provider's cost. Therefore, overbooking is common in shared environments [23], where the provider consolidates hundreds or even thousands of tenants on a single server, knowing that many tenants are likely to have long periods of low resource demands. In order to safeguard the tenant's interests in the presence of overbooking, it is important to have an objective means to establish provider **accountability**, i.e., determine if a resource assurance was not met due to the provider's fault.

Third, supporting the full richness of SQL in DaaS is important for many enterprise applications. Therefore, it is crucial that the provider **does not restrict the tenant workloads**. Further, it is also important to not make any assumptions about the workloads,

[*] Work done while visiting Microsoft Research, Redmond, WA.

Proceedings of the VLDB Endowment, Vol. 7, No. 1
Copyright 2013 VLDB Endowment 2150-8097/13/09... $ 10.00.

such as transactional vs. analysis or in-memory vs. disk-resident, or require advance knowledge of the workload.

Last, the above mechanism for ensuring a minimum resource assurance must be flexible enough to support various **policies**. On one hand, performance-sensitive line-of-business applications with continuous resource demands would benefit from guaranteed resource assurances and might be willing to subscribe to a *premium service*. On the other hand, some workloads are mostly idle with a few bursts of activity. For example, consider an online business productivity software suite, such as Office 365, which uses a DaaS to store and serve its back-end databases. Users of such applications are mostly inactive and so are the corresponding databases. Such tenants would benefit from an *economy service* with relatively small resource assurances on average over time. However, to ensure low latency response, such tenants would like their resource demands to be met quickly when the burst occurs. Therefore, resource assurances must be **elastic** and adaptable to changing workload requirements. As another example of a policy, the provider may also want to support a service class such as low-latency guarantees for OLTP workloads. Further, prices and penalties for violation may be associated with absolute assurances. In such a setting, the provider overbooking a server must be able to enforce policies that suitably trade-off its revenue and fairness among tenants.

1.2 Our approach

We presented *SQLVM* [15], an abstraction for absolute and accountable resource assurances. Each tenant is promised a **reservation** of a set of key resources—such as CPU, I/O, and memory—within a database management system (*DBMS*) which is guaranteed to be allocated provided the tenant has *sufficient demand* for the resources. Conceptually, the tenant is exposed an abstraction of a *lightweight* VM with reserved resources without incurring the overheads of classical VMs. This model of absolute resources provides a natural bridge from the on-premise settings to the cloud while being powerful enough to ensure that a tenant's performance—even the 99^{th} percentile query latency—remains unaffected in spite of multiple tenants contending for shared resources within a DBMS. Furthermore, absolute resource assurances generalize to any SQL workload without imposing any restrictions on the kinds of queries or requiring advance knowledge of the workload.

In [15], we provided an overview of SQLVM which encompasses multiple key resources within a DBMS. In this paper, we focus on the problem of effectively sharing the CPU between tenants co-located at a server each of which are executing within a SQLVM. For CPU reservations, each tenant is promised an absolute *CPU utilization* reservation which is defined as the CPU time allocated to the tenant as a percentage of the elapsed time. For instance, if a tenant T_1 is promised 30% utilization, the assurance is that if T_1 has sufficient demand for resources, the provider takes responsibility to allocate 30% CPU to T_1 irrespective of which other co-located tenants are contending for the CPU.

To establish accountability when a tenant's utilization is less than its reservation, we also present a **metering logic** for CPU, *independent* of the scheduling algorithm. Providers often overbook resources, i.e., promise a sum total of reservations exceeding the server's capacity. However, utilization lower than the reservation in an overbooked server does not necessarily imply a violation since the tenant might not have enough demand for resources. The challenge of metering is to differentiate such cases of insufficient demand from that of resource deprivation.

In spite of the rich literature on CPU scheduling and generalized processor sharing, existing approaches are not flexible to support absolute CPU reservations, be elastic, handle needs of diverse database workloads, dynamically adapt to changing resource requirements, and support provider-enforced policies with a *single* algorithm. CPU reservations or quality-of-service for CPU utilization have been proposed earlier in the context of real-time systems [13] and multimedia systems [11, 14, 16]. However, such techniques often rely on application hints (such as real-time constraints) or require workloads to have certain arrival and usage patterns (such as periodic arrival, pre-defined CPU usage behavior). Multi-tenant DBMSs serving a wide variety of workloads *cannot* make such assumptions about workload, usage, or arrival. Proportional sharing, with approaches such as lottery scheduling [24], or hierarchical scheduling approaches [8] do not provide an absolute resource assurance. To the best of our knowledge, this is the *first work that explores this problem of providing absolute CPU reservations for performance isolation in a relational DaaS without any assumptions or restrictions on the workloads*.

We delve into the intricacies of providing CPU reservations, how the CPU scheduling algorithm must adapt to support reservations, how to establish provider accountability, and how to allow providers to impose various policies that govern resource allocation. We present *largest deficit first (LDF)*, a work-conserving, dynamic priority scheduling algorithm. LDF uses a *deficit* between a tenant's CPU utilization and reservation as a means for continuous feedback about every active tenant to determine which tenant to allocate the CPU at a given context switch. We qualitatively and quantitatively show how a simple greedy heuristic and knowledge about all active tenants is important and enough to guarantee reservations. We further demonstrate how the same algorithm can be extended to support features such as elastic reservations, proportional sharing of surplus resources, and maximizing a provider's revenue.

We implemented LDF in a prototype of Microsoft SQL Azure and evaluated our design using a variety of CPU-intensive workloads and up to 400 tenants consolidated within a server. Our experiments show that even when running the server at CPU utilization greater than 95%, LDF continues to meet reservations with high accuracy, co-located tenants have minimal impact even on the 99^{th} percentile end-to-end query latency, and using LDF results in about $3\times$ to $8\times$ improvements in the 99^{th} percentile latency compared to other alternatives. Today's service providers strive to provide low variance in performance for mission critical applications where the 99^{th} percentile latency is more critical than the average or even the median [5]. Our experiments demonstrate LDF's ability to isolate the 99^{th} percentile latency even during heavy resource contention, which proves LDF's superiority in providing resource isolation. Finally, we demonstrate LDF's flexibility in adapting to a variety of workloads and policies using a single algorithm.

This paper makes the following significant contributions:

- A detailed requirements analysis and characterization of the design space for CPU sharing in a multi-tenant DaaS, and discussion of abstractions for resource isolation catering to a variety of important scenarios (§2).
- A dynamic priority scheduling algorithm to isolate tenants' resource requirements which is flexible enough to handle these requirements (§3).
- An adaptation of the algorithm to support elastic reservations to minimize query latency in high consolidation scenarios (§4) while accommodating the notions of pricing, penalties, and differentiated service quality (§5).
- An prototype implementation in a commercial multi-tenant DBMS and a detailed evaluation using a variety of workloads to demonstrates the effectiveness of the proposed approaches (§6).

This paper's contributions, and that of SQLVM, form one component towards building an end-to-end multi-tenant relational DaaS.

SQLVM enables the service provider to exercise fine-grained control on the key resources within a DBMS. Determining which tenants to co-locate can leverage approaches in workload and resource modeling [4, 25]. In the event of a performance crisis, approach such as [6] can be used for load balancing to avoid repeatedly paying penalties. Furthermore, approaches such as [26] can be used to optimize the overall provider's revenue. Note that irrespective of which technique we use for workload modeling, load balancing, or revenue optimization, once a set of tenants are co-located at a server, the SQLVM abstraction is still needed to ensure that each tenant's workload is isolated from that of others.

2. CPU SHARING PRELIMINARIES

2.1 CPU Reservations

Consider a system where multiple tenants share the same DBMS process and contend for resources at the server, a model used in DaaS services such as Microsoft SQL Azure. CPU reservation is a promise that a tenant (T_i) will be allocated a minimum CPU utilization (ResCPU_i) at the DBMS server, if the tenant has sufficient demand for CPU, irrespective of which other tenants are contending for CPU at the server. T_i's CPU utilization (CPU_i) over an interval of time (called the *metering interval*) is defined as time for which T_i's tasks use the CPU, expressed as a percentage of the total time in the interval. If T_i is promised a 10% CPU reservation at a server , it implies that in a given metering interval, T_i will be allocated 10% of the aggregated CPU time across all cores on the server. ResCPU_i can be finer than a CPU core, thus allowing the provider to consolidate more tenants than available cores, while supporting resource assurances. CPU reservations can be for a physical or a logical server. A DaaS provider with a heterogenous cluster of servers can expose a reservation on a *logical server* (similar to Amazon EC2, for instance). Once a tenant is placed at a server, its logical CPU reservation is transformed internally to the reservation on the DBMS server.

In addition to ResCPU_i, the provider might also limit T_i's maximum CPU utilization (MaxCPU_i), where $\mathit{MaxCPU}_i \geq \mathit{ResCPU}_i$. The value of MaxCPU_i used is a policy decision. However, the CPU scheduling algorithm must possess the ability to enforce this limit even if the tenant inundates the system with a lot of work.

2.2 Metering

CPU reservations do not imply static resource allocation. Therefore, it is possible that the provider allocates less CPU to a tenant, especially when resources are overbooked. A *metering logic*, which is *independently auditable*, establishes accountability when a tenant's utilization is less than its reservation. Metering differentiates low utilization due to insufficient work from that due to overbooked resources where demand exceeds capacity. Since tenants are often idle, overbooking does not necessarily imply a violation.

Metering pivots on factoring out a tenant (T_i)'s idle time, i.e., when T_i did not have CPU work and hence was not allocated the CPU. At a given scheduler, if T_i has at least one task that is either allocated the CPU or is ready to use it, then it has work that can utilize the CPU. A provider can violate a reservation by *delaying* T_i's allocation, i.e., by allocating the CPU to *another* tenant T_j even though T_i had *at least* one task ready to use the CPU. Let Delay_i denote the time T_i was delayed due to other co-located tenants using the CPU, expressed as a percentage of the metering interval length.[1] We define T_i's *effective CPU utilization* as:

$$\mathit{CPU}_i^{\mathit{Eff}} = \frac{\mathit{CPU}_i \times 100}{(\mathit{CPU}_i + \mathit{Delay}_i)} \tag{1}$$

T_i's reservation is *violated* iff $\mathit{CPU}_i^{\mathit{Eff}} < \mathit{ResCPU}_i$. The denominator corresponds to the time when the tenant was active and could potentially have used the CPU, thus factoring out idle time. Therefore, the ratio denotes T_i's effective share of the CPU in the time it was active. The multiplier 100 converts the fraction in the range $[0, 1]$ to a percentage in the range $[0, 100]$ to match the scales of ResCPU_i and $\mathit{CPU}_i^{\mathit{Eff}}$. If the ratio is greater than ResCPU_i, then during the time T_i was active, the provider allocated CPU time at a proportion greater than the reservation. Thus, T_i's lower utilization is due to insufficient demand for CPU and the provider did not violate the reservation. This definition of metering does not hold the provider accountable for the tenant being idle, while ensuring that a provider cannot arbitrarily delay a tenants task without violating the reservation. Further, Delay_i can be tracked independently at every scheduler and aggregated at the end of the metering interval.

2.3 Elastic Reservations

The CPU reservation abstraction discussed above exposes a static resource promise suitable for tenants with a steady resource demand. A static reservation is unattractive to tenants with unpredictable and bursty workloads since subscribing to a large static reservation equal to the peak demand is uneconomical, and a small static reservation equal to the average utilization limits a tenant's ability to obtain resources during workload bursts.

To make reservations economical for such *lightweight* tenants, the provider must consolidate hundreds or even thousands of such tenants on a single server. This implies that each tenant will be provided a small reservation. Tenants with workload bursts would benefit from an *elastic reservation* which allows utilization to be higher than the reservation during small bursts of activity when the tenant has high demand for resources. Elastic reservation bounds the total resource utilization for a given metering interval to the reservation promised, but allows instantaneous utilization to be driven by the tenant's resource demands. The maximum utilization at a given instant is bounded by a maximum burst size (b_i). Therefore, a mostly-inactive tenant with bursts of activity can subscribe to a small reservation (such as 0.5%) for a long metering interval (such as minutes or hours). The magnitude of the reservation depends on its average resource utilization over longer periods and the burst parameter (b_i) corresponds to its burst size which provides the tenant quick access to resources during a burst.

Metering elastic reservation is similar to that of static reservation. If T_i subscribed to an elastic reservation ResCPU_i with a burst b_i, then when T_i is active and its utilization is less than ResCPU_i for the entire metering interval, the burst is not met if $\mathit{CPU}_i^{\mathit{Eff}} < b_i$.

2.4 Revenue

CPU reservations can also be exposed directly to tenants. For instance, similar to an Infrastructure-as-a-Service (IaaS) provider, a DaaS provider can expose the CPU reservation for a tenant. When reservations are exposed directly to the tenants, it will be accompanied with a *price* which a tenant pays for the reservation [26]. Further, a reservation can also have a *penalty* which the provider refunds a fraction of the price if the reservation is violated. For instance, many of today's service providers guarantee that if service availability falls below a threshold, it will refund a fraction of the price a tenant pays. Similarly, priced CPU reservations can also be accompanied by penalty functions. Recall that a tenant T_i's reservation is violated iff $\mathit{CPU}_i^{\mathit{Eff}} < \mathit{ResCPU}_i$. The extent of violation is computed as the *fractional CPU violation*:

[1] Delay_i is different from the time spent by the tasks waiting in the runnable queue since time when T_i was allocated the CPU does not count toward Delay_i. Moreover, Delay_i only accounts for the number of CPU quanta T_i was delayed and is independent of the number of T_i's tasks that are ready to run.

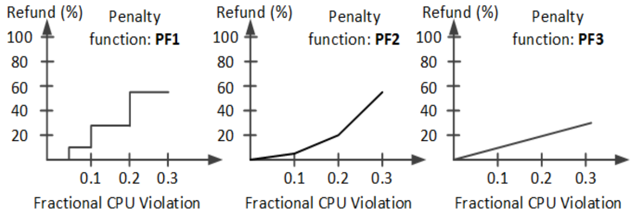


Figure 1: Penalty functions determine the fraction of the price refunded as a function of the extent of violation.

$$\mathit{ViolCPU}_i = \frac{\mathit{ResCPU}_i - \mathit{CPU}_i^{\mathit{Eff}}}{\mathit{ResCPU}_i} \tag{2}$$

The numerator is the absolute violation in the reservation and the denominator normalizes the violation allowing this metric to be used for tenants with different ResCPU_i. Different penalty functions can be associated with $\mathit{ViolCPU}_i$. Figure 1 shows a few example penalty functions which use $\mathit{ViolCPU}_i$ to determine the percentage of price refunded to the tenant. Irrespective of the function's shape, a price-aware CPU scheduling algorithm must allow the provider to maximize its revenue and minimize penalties.

2.5 Fairness

In the quest towards minimizing penalty, it is possible that in an overbooked system, certain low-paying tenants get starved of the CPU, thus making the system unfair. For instance, in the Gold, Silver, and Bronze scenario, an overbooked server with a number of active Gold tenants can deprive CPU to the Bronze tenants in order to minimize the penalties paid to the incurring violation for the higher-priced Gold tenant. We use the Jain's fairness index [10], a well-known measure used in the networking literature, to quantify such unfairness. The system is CPU utilization fair if $\mathit{ViolCPU}_i$ is the same for all tenants. Jain's fairness index is expressed as:

$$\mathcal{J}(\mathit{ViolCPU}_1, \ldots, \mathit{ViolCPU}_n) = \frac{(\sum_{i=1}^{n} \mathit{ViolCPU}_i)^2}{n \times \sum_{i=1}^{n} (\mathit{ViolCPU}_i)^2} \tag{3}$$

The value of $\mathcal{J}$ varies between 1 (all tenants have equal values of $\mathit{ViolCPU}_i$) to $1/n$ (one tenant gets the largest possible $\mathit{ViolCPU}_i$ with $\mathit{ViolCPU}_j$ being 0 for others); where n is the number of tenants with at least one task active during a metering interval. The reason we use Jain's index is that it is independent of the population size, scale or metric, is continuous, and bounded [10].

Fairness, in terms of CPU utilization, is important for the provider for customer satisfaction and retention in a competitive marketplace. Therefore, a provider may also want to balance between revenue and fairness. Jain's index can be used to evaluate the effectiveness of an price-aware scheduling algorithm that balances revenue with fairness.

3. RESERVATION-AWARE SCHEDULING

Supporting the abstraction of CPU reservations within a DBMS requires reservation-aware CPU scheduling algorithms. The dynamic nature of the tenant workloads, burstiness, different degrees of parallelism, and varying quantum lengths make it impossible to use off-the-shelf OS schedulers to support a variety of properties (as discussed in § 2) desirable from a multi-tenant DBMS.

3.1 CPU Scheduling Preliminaries

Many DBMSs directly manage a server's resources (such as CPU, memory, and disk I/O) with minimal support from the OS. Consider, for example, a DBMS running on a multi-core processor. The DBMS runs a scheduler that enforces the DBMS-specific scheduling policies; DBMS's often use a user-mode co-operative (i.e., non-preemptive) scheduler that allows more control to the system designers in determining opportune moments for yielding the CPU [22]. Our algorithms generalize to both preemptive and non-preemptive schedulers. However, our implementation uses a non-preemptive scheduler, which we use for the remaining discussion. Using a scheduler per CPU core allows the DBMS to scale to large numbers of cores. On each core, a scheduler manages a set *tasks* queued for execution on that core. A task can be in one of the following states: *running* (currently executing on the core), *runnable* (ready to execute but is waiting for its turn), or *blocked* (waiting on some resource, e.g., a lock on a data item, and hence not ready to execute). The scheduler maintains a *runnable queue* for all the runnable tasks. At a context switch boundary, the scheduler determines which task to allocate the CPU. Every task allocated the CPU uses its *quantum* of CPU which is a pre-defined maximum time in a cooperative scheduler. By having at most as many schedulers as the number of processor cores and each scheduler making at most one task runnable, the OS scheduler can schedule only those tasks made runnable by the DBMS scheduler, thus allowing the DBMS to control the CPU without relying on the OS.

3.2 The Fractional Deficit Measure

The CPU scheduler's primary goal is to meet the reservations. It is tempting to consider a solution which ensures that each tenant is provided opportunities to use the CPU which is in proportion of their ResCPU_i. For instance, an approach such as the tagging-based I/O scheduling algorithm proposed in [15] is a candidate to determine when a tenant will be allocated the CPU. However, different tenants have different quantum lengths. For instance, an OLTP workload often has shorter quanta compared to an analysis workload with CPU-intensive operators or user-defined functions. Therefore, the same number of scheduling opportunities might result in very different CPU utilization for different tenants. Varying numbers of connections and degrees of parallelism make this even more challenging. In fact, in an implementation based on scheduling opportunities, we observed significant errors in meeting reservations when OLTP and DSS workloads are co-located.

It is important that there is a continuous feedback to the scheduler based on tenant T_i's utilization (CPU_i) and its reservation (ResCPU_i). We define a metric, called the *fractional deficit* (d_i), which measures the *deficit* between T_i's utilization and reservation normalized by its reservation.

$$d_i = 1 - \frac{\mathit{CPU}_i}{\mathit{ResCPU}_i} \tag{4}$$

Incorporating the current utilization (CPU_i) and the reservation (ResCPU_i) into the scheduling decision provides feedback to the scheduler about quantum length variance and degrees of query parallelism. We normalize d_i by ResCPU_i so that when T_i uses the CPU for a quantum, the rate of decrease in d_i is inversely proportional to ResCPU_i, while if T_i was not allocated a quantum, the increase in its deficit is directly proportional to ResCPU_i. Thus, as time progresses, d_i dynamically adjusts the scheduling decisions based on CPU allocation and the reservation. Note that Equation (4) can also use $\mathit{CPU}_i^{\mathit{Eff}}$ instead of CPU_i. However, we choose CPU_i to keep the metering and scheduling logic disjoint. Later in this paper (see §5) we discuss adaptation of d_i to factor in $\mathit{CPU}_i^{\mathit{Eff}}$ and $\mathit{ViolCPU}_i$. d_i is positive if $\mathit{CPU}_i < \mathit{ResCPU}_i$, thus signifying that T_i is CPU-deficient. The scheduler computes d_i at a context switch boundary to determine how deficient T_i is and makes scheduling decisions accordingly.

There are a several interesting properties of this deficit metric which are critical to support the variety of properties discussed earlier (see §2). *First*, if the scheduler never schedules a tenant with deficit less than or equal to zero and the server is not overbooked, then the scheduler can guarantee that all reservations will be met. This property ensures that the schedulers using this metric can guarantee reservations. *Second*, even though schedulers are local to a core, T_i's CPU utilization across all cores can be computed by maintaining one shared variable updated atomically. This property provides a low cost feedback mechanism enabling the scheduler to support reservations on a multi-core processor with minimal synchronization overhead, while being able to scale to large numbers of cores. *Last*, changing ResCPU_i directly affects d_i. Therefore, using d_i for scheduling decisions allows the algorithms to almost immediately react to changes in a tenant's ResCPU_i, thus making it adaptable to changing resource and workload demands.

3.3 Deficit-based Round Robin

A scheduling algorithm, such as round robin, which does constant amount of work per context switch can scale to large numbers of concurrently-active tenants.[2] However, unfairness of round robin schedulers is well-known, especially when quantum lengths differ among tenants [18], making it unsuitable for reservations.

Inspired by Shreedhar and Varghese [18], we propose a variant of the round robin scheduler, called *deficit-based round robin (DRR)*, where the scheduler uses the deficit for the scheduling decisions. Our deficit measure (d_i) differs from that used in [18] for network packet scheduling since unlike in network scheduling where packet size is known when scheduling the packet, the amount of time a task would use the CPU is not known at scheduling time. The scheduler maintains a list of active (i.e., ready to run) tenants, called the *active list*. Each entry in the active list points to a FIFO queue of all runnable tasks of the tenant. The scheduler's runnable queue is maintained as a hash table keyed on the tenant ID. For a newly-active tenant, a new entry is added to the active list next to the current element, making enqueue a constant time operation.

The scheduler goes round robin over the active list to determine if the current tenant (T_i) can be scheduled. If $d_i > 0$ and T_i has more CPU work, then T_i is scheduled for the next quantum. If $\mathit{CPU}_i \geq \mathit{MaxCPU}_i$, then the next tenant in the active list is examined. If $d_i \leq 0$ and $\mathit{CPU}_i < \mathit{MaxCPU}_i$, then scheduling T_i is a tradeoff between the scheduling overhead and delaying allocating the CPU to T_j with $d_j > 0$. If T_i is scheduled, it is possible that there is at least one other T_j ($j \neq i$) in the active list with $d_j > 0$ which was deprived the CPU. On the other hand, if T_i is not scheduled, it increases the scheduler's cost since DRR aims to keep scheduling cost at $O(1)$ and hence inspect one (or at most a constant number of) tenants per scheduling opportunity. If T_i is not scheduled when $d_i \leq 0$, then in the worst case, if $d_j \leq 0 \; \forall j \in [1, \ldots, n]$ will result in the scheduling cost tending towards $O(n)$, where n is the number of active tenants. In our implementation, we schedule T_i once if $\mathit{CPU}_i < \mathit{MaxCPU}_i$.

While simple and efficient, the round robin nature of DRR introduces some fundamental limitations. *First*, the scheduler only has the most recent deficit of the current tenant it is inspecting. Without global information about all active tenants and their deficits, DRR needs heuristics (such as those discussed in the previous paragraph) to make a scheduling decision and ensure the cost remains $O(1)$. *Second*, when large number of tenants are active and the scheduler goes round robin scheduling a tenant at least once at every context switch, in case a highly CPU-deficient tenant appears late in the round robin order, there can be significant delays before the tenant is allocated the CPU . Such delays can result in higher variance in performance—a fact corroborated by our experiments. When striving for performance isolation, it is important to ensure low performance variance. *Last*, the round robin nature inherently limits the scheduler's ability to prioritize access. Such dynamic prioritization is critical to support features such as optimizing for revenue or providing higher priority access to tenants subscribing to more expensive service classes. Therefore, while the $O(1)$ algorithm is efficient, the feature set it can support is limited.

Algorithm 1 Largest Deficit First Scheduling (Dequeue)

```
while true do
    maxDeficit ← $-\infty$; maxDeficitTenant ← NULL
    for each active tenant $T_i$ do
        $d_i = 1 - \mathit{CPU}_i / \mathit{ResCPU}_i$
        if ($\mathit{CPU}_i < \mathit{MaxCPU}_i$ && $d_i >$ maxDeficit) then
            maxDeficit ← $d_i$; maxDeficitTenant ← $T_i$
    if (maxDeficit $> -\infty$) then
        // Select the task at the head of the queue for the tenant with largest deficit.
        ScheduleTask (maxDeficitTenant.Head)
    else
        // Either no tenant is active or all tenants have hit their maximum limit.
        ScheduleIdleQuantum ()
```

3.4 Largest Deficit First

We now present an algorithm which uses knowledge about all active tenants to select the next tenant to be allocated the CPU. We compute the deficit of all active tenants and use a greedy heuristic to allocate CPU to the tenant with the largest deficit, i.e., which is most CPU-deficient; we call this algorithm *largest deficit first (LDF)*. The scheduler maintains a per-tenant FIFO queue for tasks ready to run. A hash table keyed by the tenant ID points to a tenant's queue. Enqueue is a constant time operation involving looking up the tenant's queue and appending to it. At dequeue, the scheduler must determine the tenant with the largest deficit. To make the dequeue efficient, it might be tempting to consider a structure, such as a max heap, which will result in logarithmic dequeue complexity. However, such a structure cannot be used in this case due to a number of reasons. *First*, as noted in §3.2, a tenant's deficit changes continuously over time and at varying rates. *Second*, multiple tenants' deficits might change as a result of CPU being allocated on other cores in the system. Therefore, incrementally maintaining a heap with logarithmic complexity becomes impossible. We therefore recompute every active tenant's deficit at a context switch boundary and select the tenant with the largest deficit. Algorithm 1 presents the pseudocode of LDF's dequeue.

The LDF scheduler's enqueue is $O(1)$ and dequeue is $O(n)$, thus resulting in an overall complexity of $O(n)$, where n is the *number of active tenants at a context switch*. Note that a provider might consolidate hundreds or even thousands of tenants on a single server. However, such high consolidation is practical only because these tenants are expected to be idle most of the time; if hundreds of tenants are concurrently active, the node will already be too overloaded. Furthermore, most modern servers have tens of CPU cores. Therefore, the number of active tenants at a context switch boundary at a scheduler is typically small. Thus, even though the $O(n)$ complexity might sound theoretically daunting, the scheduling overheads are expected to be low in practice, a claim we validate in our experiments. Our experiments demonstrate that the scheduler scales to four hundred bursty tenants co-located at the same server without any noticeable overheads.

The LDF algorithm's use of the fractional deficit measure lends it several interesting properties. *First*, if a server has enough CPU capacity, LDF ensures that all the reservations can be met. Claim 1 formalizes this property of LDF.

[2] Work done by the scheduler is the sum of the work to enqueue and dequeue a task.

CLAIM 1. *If CPU capacity is not overbooked and the tenant submits sufficient work, then LDF ensures that every tenant will meet its reservation.*

PROOF. Assume for contradiction that T_i had sufficient work but was still not allocated its reservation, i.e., d_i is positive at the end of the metering interval. Since CPU was not overbooked and T_i had enough work, the only reason was T_i was deprived the CPU is that another tenant T_j was allocated a CPU share larger than its reservation. That is, T_j was allocated the CPU when d_j was negative even though d_i was positive. This is a contradiction, since the scheduling algorithm selects the tenant with the largest deficit. □

Second, LDF results in continual sharing of the CPU in proportion to the ResCPU_i, preventing one tenant with a large reservation from hogging the CPU for long periods. When T_i uses the CPU for a given quantum, the rate of decrease in d_i is inversely proportional to ResCPU_i. On the contrary, if T_i was not allocated a quantum, the increase in its deficit is directly proportional to ResCPU_i. The scheduler's goal is to meet every tenant's reservation. By selecting the tenant which is farthest from the goal (i.e., with the largest deficit), LDF ensures that the CPU bandwidth is proportionally shared in the ratio of all the active tenants' ResCPU_i. That is, if all tenants are active, T_i's proportion is: $(\mathit{ResCPU}_i)/(\sum_{j=1}^{n} \mathit{ResCPU}_j)$.

Third, the scheduler is a *dynamic priority* scheduler that is *work-conserving*. That is, a tenant's scheduling priority is determined dynamically at every context switch and unless a limit is enforced by MaxCPU_i, the scheduler can operate at 100% CPU utilization if the tenants have enough work to consume the resources.

3.5 Metering

The metering logic tracks Delay_i. Every tenant's entry in the scheduler's runnable queue contains a timestamp which is updated when a tenant transitions from idle to active. Subsequently, when a tenant T_i yields the CPU to another tenant T_j, T_i's timestamp is updated to the current time and Delay_j is incremented by the difference between the current time and the previous value of T_j's timestamp. T_i's delays are accumulated across all cores at the end of the metering interval to compute $\mathit{CPU}_i^{\mathit{Eff}}$ which is used to detect a violation in the reservation. Recall that a reservation is violated iff $\mathit{CPU}_i^{\mathit{Eff}} < \mathit{ResCPU}_i$.

4. ELASTIC RESERVATIONS

One attraction for cloud services is the elasticity of the resources. However, the basic model of CPU reservations exposes a static reservation. Recall that the fractional deficit metric (d_i) factors ResCPU_i and the scheduler's priority can be dynamically adapted by changing ResCPU_i. In this section, we discuss how this adaptable d_i metric can be used to implement an elastic reservation abstraction for tenants with bursty workloads (see §2.3).

Different shapes of the utilization curve can be achieved by internally *boosting* a tenant's reservation (and hence its priority). That is, if T_i's CPU utilization in the metering interval is less than a fraction f_i ($0 \leq f_i \leq 1$) of ResCPU_i for the entire interval, then internally, T_i's reservation is boosted to b_i, where b_i is a parameter to the scheduler. The boosted reservation is then used to compute d_i which determines T_i's priority. This boosting is internal to the scheduler to provide low response times to bursty and mostly inactive tenant workloads. The tenant's actual reservation remains unaffected and determines the total CPU allocated in the metering interval. The shape of the utilization curve depends on T_i's workload and resource demands.

The dynamic boosting algorithm introduces a couple of parameters f_i and b_i. $f_i = 0$ disables boosting and $f_i = 1$ will allow the tenant to benefit from boosting until it has exhausted its entire reservation. A value of 0.8 boosts priority until the utilization is up to 80% of the reservation and degenerates into the regular scheduling algorithm beyond that. Similarly, setting $b_i = \mathit{MaxCPU}_i$ allows the tenant's priority to be boosted to the maximum CPU utilization allowed by the system for T_i. The values used for f_i and b_i are policy decisions not addressed in this paper. Also note that this dynamic boosting is temporary, that is only for a specific scheduling opportunity.

5. PROVIDER-ENFORCED POLICIES

So far, our discussion has focused on the goal of meeting CPU reservations. We now explore adaptations to enable a provider to enforce various policies based on price, penalties, and optimize for revenue and fairness in an overbooked server. LDF adapts to cater to these additional requirements by changing the formula used to compute d_i; the rest of the scheduling logic remains unaltered.

5.1 Sharing Surplus Capacity

Once the scheduler has met all the reservations and surplus CPU capacity is available, one policy question is how to share this surplus capacity. By default, LDF shares this surplus capacity in the ratio of the ResCPU_i of all active tenants T_i. In a more general setting, the provider might want to share the CPU in the proportion of a weight (w_i) associated with T_i. w_i can correspond to the price paid, or ResCPU_i, or some other provider-determined weight. Once T_i's utilization had reached ResCPU_i, sharing in proportion of w_i is achieved by computing d_i as:

$$d_i = \frac{(\mathit{ResCPU}_i - \mathit{CPU}_i)}{w_i} \tag{5}$$

First, note that if $\mathit{CPU}_i > \mathit{ResCPU}_i$, then $d_i < 0$. Therefore, any tenant with a positive deficit continues to have higher priority and LDF still ensures that all reservations are met first. *Second*, an analysis similar to that in §3.4 shows that this modified formula for d_i proportionally shares *the surplus CPU* in the ratio $\frac{w_i}{\sum_k w_k}$ for all active tenants T_k that have met their reservation.

5.2 Maximizing Revenue

When reservations are exposed to tenants, the provider may associate a price and a penalty function (see §2). In such a setting, if a DBMS server is overbooked and there is more demand for CPU than the resources available, some reservations may be violated. When reservations have associated prices and penalties, violations may decrease the provider's revenue. It is natural for the provider to want to maximize its revenue. Since the basic LDF algorithm is unaware of prices and penalties and does not optimize for revenue, it can result in considerable penalties in an overbooked server. Penalties can be minimized (and revenue maximized) by allocating the CPU to the tenant with the highest potential penalty if deprived the CPU. This greedy heuristic can also be added to LDF by modifying deficit computation as follows:

$$d_i = \mathbf{PF}_i(\mathit{ViolCPU}_i) \times \mathit{price}_i \tag{6}$$

The scheduler computes T_i's fractional violation ($\mathit{ViolCPU}_i$) and uses T_i's penalty function ($\mathbf{PF}_i$) to obtain the fractional penalty expected for this metering interval assuming T_i continues to receive the same CPU proportion. When multiplied by price_i that T_i pays, d_i is the absolute penalty which the provider is liable to T_i. Thus, LDF will select the tenant with the largest potential for penalty. Note that for $\mathit{CPU}_i \geq \mathit{ResCPU}_i$, penalty is zero and so is d_i. Equations (4) or (5) can be used for setting the value of d_i, which will be negative in this case. Also note that simply replacing CPU_i by

CPU_i^{Eff} in Equation (4) is also not enough as it does not account for the penalty function's shape or the price paid.

5.3 Revenue and Fairness

The above mentioned greedy heuristic can deprive CPU to some tenants more than others, thus making the system unfair in terms of utilization fairness. While fairness might not be most critical for the provider, it is important to prevent complete starvation and considerable performance degradation for some tenants. Fairness is also important for customer satisfaction and retention in a competitive marketplace. When using the deficit formula from Equation (4), LDF is utilization fair while using Equation (6) makes it completely revenue oriented. The scheduler can trade-off fairness and revenue by combining both factors into the deficit computation:

$$d_i = \alpha \times \underbrace{\left(1 - \frac{CPU_i}{ResCPU_i}\right)}_{utilization\,fairness} + (1 - \alpha) \times \underbrace{\left(\frac{\mathbf{PF}_i(ViolCPU_i) \times price_i}{\mathbf{Max}(price_i)}\right)}_{maximizing\ revenue} \quad (7)$$

The first component (from Equation (4)) contributes to utilization fairness while the second component (from Equation (6)) maximizes revenue. To match with utilization fairness component which is in the range $[0, 1]$ for a CPU-deficient tenant, we normalize the revenue component by dividing $price_i$ with the maximum price among all tenants. A weighted sum of these two factors allows the provider to tune between utilization fairness and revenue. The tuning parameter α is the *fairness index* which determines how fair the scheduler is: $\alpha = 0$ configures the scheduler to maximize for revenue and $\alpha = 1$ configures it to be fair. Note that α does not directly correspond to any fairness measure, such as the Jain's index. α can be configured statically based on the system's policy or set dynamically to achieve a target value fairness measure. Our experiments show how statically setting α affects revenue and fairness; the latter is left for future work.

5.4 Discussion

In this section, we presented three extensions to the LDF scheduler by modifying how a tenant's fractional deficit is computed. Algorithm 2 shows the pseudocode of the LDF scheduler's dequeue function which combines the three extensions along with boosting. *Thus, a single algorithm is flexible enough to handle a variety of requirements desirable for a CPU scheduler of a multi-tenant DaaS.*

LDF generalizes to I/O and network sharing as well. For instance, assume a tenant T_i is promised an I/O reservation of 2Mbps. d_i needs to be adapted to consider T_i's I/O utilization (computed by accumulating the size of T_i's I/Os) and replacing the denominator by the I/O reservation. It is straightforward to show that by scheduling the tenant with the largest I/O deficit, LDF provides the same behavior as for CPU. Implementing LDF for I/O and network sharing is left as future work.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup and Baselines

We implemented our proposed CPU reservations abstraction, metering, and the different scheduling algorithms in a prototype of Microsoft SQL Azure. This section experimentally evaluates the effectiveness of these abstractions and algorithms using a variety of workloads in a multi-tenant setting. Our evaluation uses a workload suite comprising four different workloads that represent diverse resource and performance requirements: TPC-C and Dell DVD Store[3] benchmarks are OLTP-style workloads; TPC-H benchmark is a DSS-style workload; and a synthetic micro-benchmark (called CPUIO) that generates queries that are CPU- and/or I/O-intensive.

Algorithm 2 Dequeue operation of the extended LDF Scheduler

```
while true do
    maxDeficit ← $-\infty$; maxDeficitTenant ← NULL
    for each active tenant $T_i$ do
        reservation ← $ResCPU_i$; schedOpt ← GetSchedulerOption()
        if (schedOpt = BOOSTED & IsElasticRes ($T_i$) & $CPU_i < f_i \times ResCPU_i$)
        then
            reservation ← $b_i$
        $d_i \rightarrow 1 - CPU_i/$reservation
        if ($CPU_i > ResCPU_i$) then
            // Proportionally sharing the surplus CPU capacity.
            $d_i \leftarrow (ResCPU_i - CPU_i)/w_i$
        else if ((schedOpt = MAXIMIZE_REVENUE)) then
            $d_i \leftarrow \mathbf{PF}_i(ViolCPU_i) \times price_i$
        else if (schedOpt = REVENUE_AND_FAIRNESS) then
            $d_i \leftarrow \alpha \times \left(1 - \frac{CPU_i}{ResCPU_i}\right) + (1 - \alpha) \times \left(\frac{\mathbf{PF}_i(ViolCPU_i) \times price_i}{\mathbf{Max}(price_i)}\right)$
        if ($CPU_i < MaxCPU_i$ && $d_i >$ maxDeficit) then
            maxDeficit ← $d_i$; maxDeficitTenant ← $T_i$
    Schedule tenant with largest deficit.
```

The TPC-C workload contains a mix of read/write transactions portraying a wholesale supplier. The Dell DVD Store benchmark emulates an e-commerce workload where transactions represent user interactions with the web site. The TPC-H benchmark simulates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The CPUIO benchmark comprises of a single table with a clustered index on the primary key and a non-clustered index on a secondary key. The workload consists of three query types: (i) a CPU-intensive computation; (ii) a query involving a sequential scan with a range predicate on the primary key of the table; and (iii) a query with a predicate on the non-clustered index which performs random accesses to the database pages.

Each tenant is an instance of one of these workloads and connects to its own logical database. Tenants are hosted within a single instance of the database server with a 12 core processor (24 logical cores), 96GB memory, data files striped across three HDDs, and transaction log stored in an SSD. The tenants' database sizes range from 500MB to 5GB.

An experiment starts with loading a number of tenant databases, warming up the cache by running all the workloads concurrently for 30 minutes, followed by another 30 minute period where all the measurements are collected, aggregated, and reported. In all experiments, the combined tenant CPU utilization is limited to 95% to leave head room for system management and maintenance tasks. Since the focus is on CPU, the warmup period loads the working set in memory so that very little or no I/O is incurred during the experiment. Further, we use an I/O reservation technique, described in [15], to isolate I/O requirements of the tenants, if any. We evaluated two configurations: one where all tenants execute the exact same workload and another when a tenant's workload is chosen randomly from the workload suite. In these experiments, we do not consider the case of tenants joining or leaving; we refer the interested readers to [15] for such experiments.

We compare LDF against multiple baselines. *First*, the deficit round robin algorithm (DRR) (§3.3). *Second*, an adaptation of the earliest deadline first (*EDF*) algorithm [13]. EDF was designed for real-time applications with deadlines. It might be tempting to use the end of the metering interval as the deadline. However, such a deadline is not enough since all tenants would then have the same deadline. We therefore use a variant of EDF used in the Atropos scheduler of Xen [12] where a tenant T_i's deadline is set to the actual interval's deadline minus the tenant's deficit ($ResCPU_i - CPU_i$). *Third*, a system without any CPU reservations (*No Res*) which provides best effort allocation without any

[3] http://linux.dell.com/dvdstore/

Table 1: Tenant T_8's average CPU utilization.

Num	LDF		DRR		EDF		No Res	
Bullies	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
0	25	1.06	25.06	0.71	25.09	0.84	11.89	1.29
2	25.11	1.49	23.86	2.24	25.14	1.39	9.48	0.78
4	25.12	1.61	20.93	3.89	25.11	1.46	7.93	0.55
6	25.09	1.7	19.36	4.9	25.12	3.45	6.82	0.49
8	25.09	1.62	17.5	6.01	24.85	3.48	5.97	0.42

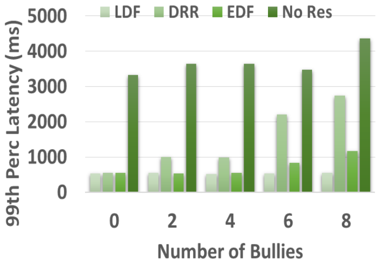


Figure 2: 99^{th} percentile latency for one of T_8's query types.

prioritization. Note that LDF and DRR use the same deficit measure but have different underlying strategies—LDF is greedy while DRR is round robin. On the other hand, LDF and EDF are both greedy, but use different metrics for making the greedy choice.

6.2 Over-provisioned Server

In this experiment, we use an over-provisioned server, i.e., where available resources is greater than the aggregate of reservations. We use two classes of tenants in this experiment. The first class comprises tenants with CPU reservations while the second class is without reservations. All tenants, irrespective of their class, generate CPU-intensive workloads, resulting in heavy CPU contention at the DBMS server. The eight tenants with reservations ($T_1, \ldots, T_8$) are of interest for performance and resource isolation. These tenants have CPU reservation of 5%, 8%, 8%, 8%, 10%, 10%, 10%, and 25%, resulting in 85% capacity reservation. The goal of the second class of tenants is to hog resources and generate contention, hence we call them resource hogs or *bullies*. The number of bullies is varied from zero to eight. Since bullies do not have any reservation, the server is over-provisioned in terms of reservations, even with eight bullies. For brevity, as a representative, we report the utilization and latency numbers for one of the tenants (T_8) and when all tenants (including the bullies) are executing the same workload; similar trends were observed for other tenants and configurations.

Table 1 shows the average CPU utilization of tenant T_8; CPU utilization was sampled every second and averaged over the duration of the experiment. Recall that *ResCPU*$_8$ was 25% for LDF, DRR, and EDF. When there are no bullies, all of LDF, DRR, and EDF result in comparable utilization. However, as more bullies are introduced, *only LDF continues to meet the reservation with very little variance.* DRR's round robin nature implies that the bully workloads are scheduled at least once in every round through the runnable queue. Hence, T_8's (as well as that of $T_1, \ldots, T_7$) utilization continues to decrease as more bullies are added. EDF's greedy heuristic enables it to meet all the reservations, a property shared with LDF. However, EDF's standard deviation is higher than LDF. This is because LDF results in continual proportional sharing of the CPU (see §3.4). On the other hand, EDF results in more coarse-grained sharing, since the deadlines are dominated by the absolute values of *ResCPU*$_i$. Different from LDF, DRR, and EDF, No Res does not promise any minimum resource allocation, thus resulting is more drastic impact on resource allocations as bullies are added.

Table 2: End-to-end latency (in ms) for one of T_8's query types.

Num	LDF		DRR		EDF		No Res	
Bullies	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
0	331	77	278	100	336	84	379	1115
2	350	76	236	327	339	83	405	1156
4	332	97	255	462	341	98	386	1212
6	336	74	283	559	359	204	400	1447
8	341	89	304	777	368	231	501	1505

The benefits of good resource isolation is also evident from the end-to-end latency for one of the query types of T_8. Figure 2 reports the 99^{th} percentile latency observed by the tenant. As is evident, T_8's latencies remain unchanged as the number of bullies are increased. Since EDF's CPU isolation is better compared to DRR, so is it's 99^{th} percentile latency. However, LDF's lower variance in allocation implies best isolation of the 99^{th} percentile latency compared to all other algorithms. With eight bullies, *T_8's 99^{th} percentile latency when using LDF is $\sim 5\times$ compared to DRR, $\sim 2\times$ better compared to EDF, and $\sim 8\times$ better compared to No Res.* Service providers strive to provide low variance in performance for mission critical applications where 99^{th} percentile latency is much more critical compared to the average or the median latency [5]. Therefore, LDF's ability to isolate even the 99^{th} percentile latency in scenarios of heavy resource contention validates our claim of performance isolation.

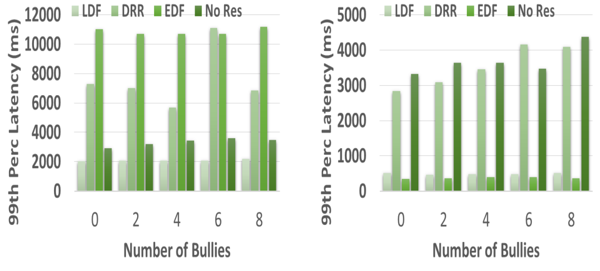
While our focus is on the 99^{th} percentile, we observed an interesting behavior when analyzing the average latency and the variance which tenants observe. Table 2 reports the average latency (in ms) and the standard deviation for the same query. There are two important observations: *(1)* DRR's average latency (Column 4) is lower compared all other techniques (columns 2, 6, and 8). However, due to the high variance observed in performance, DRR's 99^{th} percentile is worse (see Figure 2). Even though LDF's average is higher than that of DRR, the low variance explains why LDF's 99^{th} percentile latency is the lowest; and *(2)* neither the average nor the standard deviation increase for LDF as the number of bullies are increased, which further asserts LDF's ability to effectively isolate resources and performance. Therefore, *using LDF results in negligible performance impact on a tenants workload due to other co-located workloads.* This is remarkable considering that the server was running at 85% to 98% average utilization in all the experiments. The lower average latency and higher variance in DRR results from a combination of the round robin nature and accumulation of deficits. As discussed earlier (§3.3), when many tenants are concurrently active, there can be considerable delays between successive rounds when the scheduler inspects T_i. For queries arriving in the intervening period, the wait times will be higher. However, if T_i's accumulated deficit is also large, it implies that when T_i is scheduled, it may receive a burst of CPU allocation if it has enough work. Queries arriving during this burst will observe low wait times if they complete within this burst. This bimodal distribution is the reason why DRR results in a lower average latency but higher variance.

6.3 Overbooking and Violations

The setup is similar to that in the previous experiment in an over-provisioned server, with one difference: the reservations for $T_1, \ldots, T_8$ are each doubled, i.e., the reservations are 10%, 16%, 16%, 16%, 20%, 20%, 20%, and 50% respectively, and so are the maximums for max only. The reservations add up to 170% and hence the server is overbooked. Similar to the previous experiment, the number of bullies is varied from zero to eight.

Table 3: CPU utilization and metering with no bullies.

Tenant	LDF				DRR				EDF			
	CPU_i		% Violation		CPU_i		% Violation		CPU_i		% Violation	
	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
T_1	5.75	0.37	2.94	0.4	6.35	2.51	1.36	1.95	0.93	0.53	8.54	0.97
T_3	9.09	0.62	4.91	0.64	9.58	3.81	2.67	3.26	6.89	0.58	8.18	1.0
T_5	11.31	0.89	6.47	0.97	11.69	4.9	3.69	4.22	10.87	0.68	8.04	1.13
T_8	27.89	2.45	17.68	2.32	26.67	12.58	12.97	9.97	40.81	3.01	7.48	1.37



(a) T_1

(b) T_8

Figure 3: 99^{th} percentile latency observed for one query type of T_1 and T_8 in an overbooked server.

Table 4: T_8's CPU utilization with varying number of bullies.

Num Bullies	LDF		DRR		EDF		No Res	
	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.	Avg.	Std. Dev.
0	27.89	2.45	26.67	12.58	40.81	3.01	11.89	1.29
2	27.89	1.75	24.32	11.29	40.67	3.19	9.48	0.78
4	27.70	2.86	28.24	9.99	40.5	2.95	7.93	0.55
6	27.64	1.43	17.71	12.28	40.58	2.59	6.82	0.49
8	27.47	2.55	14.32	11.035	40.56	2.96	5.97	0.42

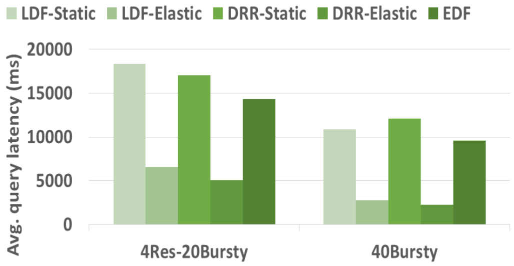


Figure 4: Average latency for a tenant with bursty workload subscribing to static and elastic reservations.

Table 3 reports the average and standard deviation of CPU_i and the percentage violation for all the tenants in an experiment without any bully workloads. Even when the server is overbooked, *LDF continues to allocate CPU at a steady rate with low variance.* Recall that in an overbooked setting, LDF proportionally shares the resources among the active tenants where a tenant's share is $\frac{ResCPU_i}{\sum ResCPU_i}$ of the available CPU capacity of the server. Since we limit the aggregate server utilization to 95%, T_1's share is 5.58%, $T_2 - T_4$'s share is 8.94%, $T_5 - T_7$'s share is 11.18%, and T_8's share is 27.94%. As is evident, LDF results in allocation which closely matches the tenant's proportion. DRR follows closely, but the allocation differs slightly from the proportion. However, similar to the previous experiment, the variance is much higher for DRR. On the other hand, EDF favors tenants with larger reservations. At the start of the metering interval, T_8 has the largest reservation and hence has the earliest deadline, a condition which remains true until $CPU_8 = ResCPU_8 - ResCPU_7$, where T_7 has the second largest reservation. Beyond this point, T_8 and T_7's allocation alternate until their deadline equals that of T_5, the tenant with the next highest reservation. This behavior of EDF stems from using the absolute value of $ResCPU_i$ in computing the deadline. LDF's use of fractional measures for the deficit results in finer-grained sharing during the entire metering interval. This unfairness is magnified in an overbooked server since earlier deadlines of tenants with large reservations might deprive tenants with smaller reservations and hence later deadlines. Measured quantitatively, in this experiment, using LDF results in the Jain's index score of $\mathcal{J} = 0.99$ while EDF results in $\mathcal{J} = 0.87$.

This unfairness is also evident in the end-to-end query latencies. Figure 3 plots the 99^{th} percentile latency for the same query executed by T_1 (smallest reservation) and T_8 (largest reservation). There are two important observations. *First*, the greedy nature both LDF and EDF allows them to dynamically priority the tenants with reservations over the bullies, thus resulting in considerably better performance isolation when compared to DRR and No Res. This fact is further affirmed by Table 4 which reports T_8's average CPU utilization. As is evident, T_8's CPU allocation degrades considerably as more bullies are added (see columns 4 and 6). It is remarkable that *even on an overbooked server hosting sixteen resource-intensive tenants, the 99^{th} percentile latency with LDF remains unaffected as the number of bullies and the resource contention increases.* Compared to DRR and No Res, tenants observe between 6× to 8× lower latency for the 99^{th} percentile. The average query latencies show a pattern similar to that of the previous experiments and the results are omitted for brevity. *Second*, the unfairness of EDF results in better end-to-end performance for T_8 (which is allocated more CPU in EDF that with LDF; see Table 3 last row) at the expense of considerably worse performance for T_1 which is almost deprived of CPU (see Table 3 first row). Note that, LDF continues to deliver excellent isolation irrespective of the magnitude of the tenant's reservation.

Another important observation from Table 3 is that the metering logic detects violations, thus establishing accountability when the provider fails to allocate resources in an overbooked scenario. Since metering logic factors out time when the tenant is not active, utilization and percentage violation need not add up to $ResCPU_i$. Note that when LDF uses the deficit measure of Equation (4), its goal is to be utilization fair. Hence, tenants with larger reservation will observe larger violations. DRR's lower violation compared to LDF is again due to bursty CPU allocation due to the round robin nature. On the other hand, EDF prefers tenants with larger reservation, hence resulting in lower violation for such tenants.

6.4 Elastic Reservations

In this section, we evaluate the effectiveness of elastic reservations for bursty workloads. In this experiment, multiple bursty workloads are co-located on the same server. A bursty tenant generates a burst of CPU activity where the burst size and the interval are configurable. We configure the workload to generate at least one burst every minute with the burst size set to use up to 5% of

CPU during the period of the burst. Over a long period of time, a bursty tenant's average utilization is 0.5%.

We evaluate two different scenarios: (i) four resource-intensive static reservation tenants are co-located with twenty bursty tenants (denoted as *4Res-20Bursty*); and (ii) forty bursty tenants are co-located with no other resource-intensive tenants (denoted as *40Bursty*). In one setup, the bursty tenants are given a *static* reservation of 1% of the CPU metered every second (enough to meet their average CPU requirement of 0.5%) and in another setup, the bursty tenants are given an *elastic* reservation of 1% metered every minute with $b_i = 30\%$ and $f_i = 0.8$. In both scenarios, the server is over-provisioned.

Figure 4 plots the average end-to-end query latency as observed by the bursty tenants (averaged over the average latency of each bursty tenant) with static and elastic reservations for the bursty workloads. As is evident, an elastic reservation, for both LDF and DRR, results in 3× to 5× reduction in the query latency. Further, in the *4Res-20Bursty* scenario, this boosting result in negligible (less than 1% measured) impact on the average and 99^{th} percentile latency of the resource intensive tenants with static reservations. Therefore, elastic reservations result in significant latency improvements for bursty workloads without much overhead on any co-located static reservation tenants. EDF's deadlines are not cognizant of elastic reservations, and hence the latencies are comparable to static reservations in LDF and DRR.

6.5 Revenue and Fairness

In this section, we evaluate the LDF scheduler's ability to maximize the provider's revenue and in trading off revenue with utilization fairness. The first experiment focuses on optimizing revenue (using the deficit measure of Equation 6) while the second experiment evaluates the impact of the fairness index (α) on revenue and fairness (using the deficit measure from Equation 7).

In this experiment, we consider a scenario where there are three service classes: Gold ($ResCPU_i$ = 32%, price 20 cents), Silver ($ResCPU_i$ = 16%, price 10 cents), and Bronze ($ResCPU_i$ = 8%, price 5 cents). In addition, every class has an associated penalty function with a higher-priced class having a more stringent function, i.e., a higher fractional penalty for the same fractional violation. Penalty functions are modeled as step functions with varying slope. We run a set up with eight co-located tenants with reservations. One configuration ($C1$) co-locates three Gold, three Silver, and two Bronze tenants with an aggregate CPU reservation of 170% and a maximum possible revenue of 100 cents. The second configuration ($C2$) co-located one Gold, four Silver, and three Bronze tenants with a total CPU reservation of 120% and a maximum possible revenue of 75 cents. In both configurations, we run resource-intensive workloads such that resource demands are higher than capacity. That is, violations are inevitable and the goal is to minimize the penalty.

Figure 5 plots the results for both configurations ($C1$ and $C2$) in a setting without bullies and one with eight resource bullies. In each sub figure, each group of bars correspond to the LDF and DRR schedulers running without optimization (*-NoOpt*) and with revenue optimization (*-Opt*) and the EDF scheduler. Figure 5(a) plots the % penalty, i.e., the percentage of the maximum possible revenue refunded as penalty for violating reservations, as the vertical axis. As is evident, LDF optimized for revenue results in considerably lesser penalty when compared to the un-optimized version. Furthermore, as expected, $C2$ results in lower penalty since it is less overbooked and hence has lesser probability of penalties. Even though DRR uses the same deficit measure, the round robin nature does not provide DRR knowledge of deficits of all active tenants and in most cases, local decisions do not result in the desired global outcome of maximizing revenue. EDF uses a greedy heuristic and favors tenants with larger reservations (and hence higher potential penalties for the same violation), its penalties are lower than LDF. However, EDF is also unaware of the price, the violation incurred, and the shape of the penalty function. Therefore, LDF's revenue-optimized version results in considerably lower penalties. Figure 5(b) plots the Jain's fairness score along the vertical axis. As expected, higher revenue of the optimized scheduler comes at the cost of lower fairness, i.e., by providing more resources to the tenants subscribing to the higher-priced service class. Finally, providing preferential treatment to the higher-priced tenant implies better performance for the tenants. Figure 5(c) reports the 99^{th} percentile latency for one query type of T_8 which is in the Gold class in both configurations. As is evident, LDF-Opt results in 20% to 30% improvement in the 99^{th} percentile latency. Therefore, the optimized LDF scheduler provides improved service quality to tenants paying for higher service classes as well as maximizes provider revenue.

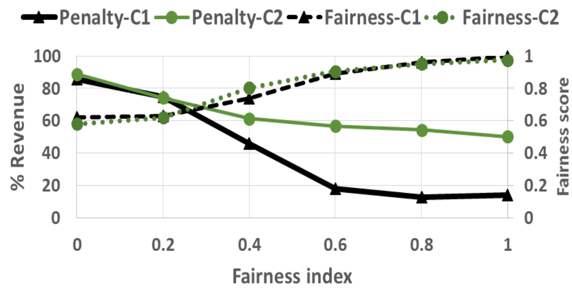


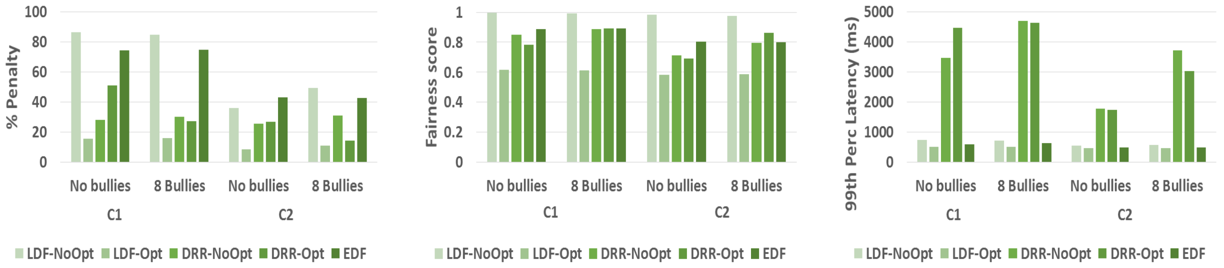
Figure 6: Trading-off provider's revenue with utilization fairness using the fairness index (α).

Finally, Figure 6 demonstrates the LDF scheduler's ability to balance between revenue and fairness by exposing a fairness index (α) which the provider can set. The horizontal axis plots the different values of α, the primary vertical axis plots the provider's revenue as a percentage of the maximum possible revenue (100 for $C1$ and 75 for $C2$), and the secondary vertical axis plots the fairness score (Jain's index). The solid lines plot the revenue, the dotted lines plot the fairness score. The thicker lines with triangular markers correspond to $C1$ and the thinner lines with circular markers correspond to $C2$. We use a setup similar to the previous experiment and vary α from 0 (least fair, most revenue optimized) to 1 (most fair, least revenue optimized). As is evident, as we increase the fairness index, the revenue decreases and fairness increases for both configurations. Thus, LDF allows the provider to select a balance that suits its requirements and business needs. Therefore, not only does LDF result in better tenant experience, it allows the provide to exercise fine-grained control and enforce various higher-level policies to suit a diverse set of requirements.

6.6 Scheduling overhead

Earlier, we noted that even though LDF's scheduling cost is linear to the number of tenants active at a given context switch, LDF's scheduling overhead is expected to be low in practice. These experiments compare the scheduling overheads of LDF to DRR and a scheduler without CPU reservations.

In one setup, we co-locate hundred bursty tenants and compare the overheads running with the LDF and the DRR schedulers. Each bursty tenant generated a burst of CPU activity at least once every twenty seconds. The DBMS server was using about 40% CPU on average during the experiment. The performance overhead of



(a) Penalty as % of subscription.

(b) Fairness (Jain's index).

(c) 99^{th} percentile latency of T_8.

Figure 5: Differentiated service and optimizing for provider revenue.

LDF when compared to DRR, measured in terms of query latency and throughput as observed by the tenants, was low (between 1% to 3%). In another setup, we ran up to forty resource-intensive tenants and compared the performance of LDF with a best-effort scheduler. The overheads were negligible (less than 3% decrease in the throughout aggregated across all tenants). In both cases, the overheads fade in light of LDF's rich feature set. To further demonstrate LDF's scalability, we also ran an experiment with four hundred mostly inactive tenants generating periodic bursts of activity. There was no measurable increase in the scheduler's overheads.

In an adversarial scenario where forty tenants each with fifty concurrent connections (i.e., an aggregate of 2000 connections) were issuing queries concurrently on all connections without any think time, we observed about 5% - 10% decrease in aggregate throughput with the LDF scheduler when compared to a best effort scheduler that does not provide *any* assurances. While this scenario will be infrequent in practice, this experiment confirms that LDF's overheads remain manageable even under heavy load. Therefore, LDF provides flexibility and assurances paramount in a multi-tenant setting with minimal overheads in practice.

6.7 Summary

Experiments presented in this section using a variety of workloads and a wide variety of scenarios validate LDF's ability to provide excellent performance isolation while being flexible enough to cater to various tenant's requirements and provider's policies. Results in this section demonstrate that:

- LDF meets reservations with high accuracy when enough resources are available, which in turn results in excellent performance isolation as reflected by the negligible effect of other tenants' workload on a tenant's 99^{th} percentile latency;
- in an overbooked server, LDF continues to isolate resources irrespective of the reservation's magnitude and detects violations to establish accountability;
- LDF can effectively support elastic reservations for bursty workloads to lower tenant's average latency by $\sim 3\times$ to $5\times$, thus improving the tenant's experience;
- LDF is able to support higher-level provider-enforced policies such as optimizing for revenue and fairness; and
- LDF introduces minimal overheads when compared to DRR or a scheduler providing no assurances.

7. RELATED WORK

Classical approaches of fair sharing, used in many OS and network packet schedulers [18], are not flexible enough to support service differentiation and absolute assurances critical in multi-tenant systems. Static priorities, used in OS schedulers and many commercial DBMSs, and proportional sharing [8,20,24] provide coarse service differentiation but do not provide absolute assurances. On the contrary, CPU reservations expose an absolute resource assurance. Static CPU reservations has also been explored in real-time [11] and multimedia systems [14] where the goal was only to support low latency response to time-critical applications. Absolute reservations with accountability differentiates our work.

Various CPU scheduling algorithms have also been proposed in the literature. One class of schedulers target proportional sharing. Examples include lottery and stride-based scheduling [24], Stoica et al. [20], Goyal et al. [8], and Chandra et al. [1]. As discussed in Stoica et al. [21], resource reservations and proportional sharing are dual problems. In theory, a proportional sharing algorithm can be used to support reservations [19]. However, in the presence of workload variety, such as different CPU quantum lengths, tenants joining and leaving, and varying resource demands, guaranteeing a reservation requires a complex feedback-driven controller which continuously adapts the tenants' proportions. In contrast, LDF provides a simple approach to guarantee reservations relying on the feedback of the deficit measure while supporting many properties similar to proportional sharing schedulers (see §3.4), thus providing the best of both worlds.

Another class of algorithms was proposed to support reservations or low latency characteristics in real-time and multimedia applications; the earliest proposals were rate monotonic (RM) scheduling and earliest deadline first (EDF) [13]. EDF is a dynamic priority algorithm with a greedy heuristic similar to that of LDF; RM on the other hand is a static priority scheme. Both algorithms rely on a notion of rate or deadlines common in real-time systems. Such assumptions about predictable request arrival rates or deadlines declared upfront are impractical in a multi-tenant DBMS serving a variety of workloads. The EDF scheduler has been adapted in various other scenarios for providing real-time assurances or CPU reservations [2, 3, 12]. Jones et al. [11] rely on CPU reservations and application-specified time constraints to pre-compute a scheduling graph which determines how tasks are scheduled. However, such pre-computation of the schedule limits the applicability of the algorithm in dynamic environments which we consider. SMART [16] uses a combination of static priorities and virtual time to achieve proportional sharing and meet time constraints. Similarly, Mercer et al. [14] support CPU reservations relying on application cooperation and an estimate of the CPU times from the programmers to map an application's quality-of-service (QoS) requirements into resource requirements. Again, such application hints are impractical in a multi-tenant DBMS. On the contrary, LDF does not require any application hints, assumptions about the workload or its arrival patterns, or impose any limitations on the workloads supported.

Regehr and Stankovic [17] present a mechanism to adapt a real-time scheduler, similar to EDF, to account for time spent in Kernel mode. Similarly, Gupta et al. [9] propose a technique to isolate the and determine a tenant's (a VM in the paper) CPU utilization in the user domain and that consumed by the hypervisor when performing work on behalf of the VM. Our work focuses on isolating the CPU consumption between the tenants. Govindan et al. [7] present a communication-aware CPU scheduler that prioritizes communication-oriented applications over CPU-intensive ones. Variants of LDF which give preferential treatment to bursty workloads or higher-paying tenants are, in principle, similar to [7].

8. CONCLUDING REMARKS

Resource sharing is inevitable in a multi-tenant DaaS, causing concerns that a tenant's performance may be impacted due to resource contention with other co-located workloads. We focused on the problem of effective CPU sharing among tenants co-located at a server. We presented the abstraction of CPU reservations which provides an absolute assurance of resources to tenants without restricting the tenant workloads by any form or means. We presented two variations of the CPU reservations model—a static reservation and an elastic reservations—which are targeted towards tenants with very different resource and performance requirements. We also presented a metering logic to establish provider accountability. Finally, we presented a single scheduling algorithm which is flexible enough to meet reservations and provide fine-grained control on resource while catering to a variety of tenant workloads and provider-enforced policies. Implementation in a prototype of SQL Azure and thorough evaluation demonstrated the effectiveness of the proposed CPU sharing abstractions and the LDF scheduler.

Acknowledgements

The authors would like to thank Pamela Bhattacharya, Surajit Chaudhuri, Christian Konig, and the anonymous reviewers for useful feedback to improve this paper. Several members of the Microsoft SQL Azure group, including Morgan Oslake, George Reynya, and Leigh Stewart, have provided feedback that influenced our work.

9. REFERENCES

[1] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: a proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI*, pages 45–58, 2000.

[2] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[3] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi. Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling. In *Euro-Par Workshops*, pages 657–664, 2010.

[4] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, pages 313–324, 2011.

[5] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[6] A. Elmore, S. Das, A. Pucher, D. Agrawal, A. E. Abbadi, and X. Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD*, pages 517–528, 2013.

[7] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms. In *VEE*, pages 126–136, 2007.

[8] P. Goyal, X. Guo, and H. M. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *OSDI*, pages 107–121, 1996.

[9] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Middleware*, pages 342–362, 2006.

[10] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. Technical report, DEC, September 1984.

[11] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *SOSP*, pages 198–211, 1997.

[12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *J. of Sel. areas in Comm.*, 14(7):1280–1297, 1996.

[13] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973.

[14] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *ICMCS*, pages 90–99, 1994.

[15] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*, 2013.

[16] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *SOSP*, pages 184–197, 1997.

[17] J. Regehr and J. A. Stankovic. Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems. In *RTAS*, 2001.

[18] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375 –385, 1996.

[19] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *OSDI*, pages 145–158, 1999.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS*, pages 288 –299, 1996.

[21] I. Stoica, H. Adbel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Mult. Comp. and Networking*, pages 207–214, 1997.

[22] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[23] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource overbooking and application profiling in a shared Internet hosting platform. *ACM Trans. Internet Techn.*, 9(1), 2009.

[24] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, MIT, 1995.

[25] P. Wong, Z. He, and E. Lo. Parallel Analytics as a Service. In *SIGMOD*, pages 25–36, 2013.

[26] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *SoCC*, 2011.