MSR-TR-2018-5
DC-DRF: Equitable Multi-resource Sharing at Public Cloud Scale

Ian A. Kash
University of Illinois at Chicago
iankash@uic.edu

Greg O'Shea
Microsoft Research
gregos@microsoft.com

Stavros Volos
Microsoft Research
svolos@microsoft.com

ABSTRACT

Public cloud datacenters implement a distributed computing environment built for economy at scale, with hundreds of thousands of compute and storage servers and a large population of predominantly small customers often densely packed to a compute server. Several recent contributions have investigated how equitable sharing and differentiated services can be achieved in this multi-resource environment, using the Extended Dominant Resource Fairness (EDRF) algorithm. However, we find that EDRF requires prohibitive execution time when employed at datacenter scale due to its iterative nature and polynomial time complexity; its closed-form expression does not alter its asymptotic complexity. In response, we propose Deadline-Constrained DRF, or DC-DRF, an adaptive approximation of EDRF designed to support centralized multi-resource allocation at datacenter scale in bounded time. The approximation introduces error which can be reduced using a high-performance implementation, drawing on parallelization techniques from the field of High-Performance Computing and vector arithmetic instructions available in modern server processors. We evaluate DC-DRF at scales that exceed those previously reported by several orders of magnitude, calculating resource allocations for one million predominantly small tenants and one million resources, in seconds. Our parallel implementation preserves the properties of EDRF up to a small error, and empirical results show that the error introduced by approximation is insignificant for practical purposes.

1. INTRODUCTION

This is an extended edition of a paper first presented at the ACM Symposium on Cloud Computing 2018 [27].

Public cloud datacenters (DC), of the sort hosting enterprise customer workloads, face a fundamental trade-off between providing *performance isolation* for each customer and achieving high *resource utilization*. Provisioning for each customer's peak demand for any given resource necessarily leaves resources idle much of the time, while oversubscribing resources runs the risk of one customer affecting another. Each customer has a set of virtual resources, collectively called a *tenant*, which may include one or several virtual machines (VMs), while the underlying physical resources are shared between tenants. Shared resources in a modern datacenter typically include storage servers [11, 18, 23, 29, 42], software load balancers [33, 40, 46], middle-boxes [5, 16], shared caches [16, 41], bump-in-the-wire offload devices [12], and the datacenter network [7, 8, 21, 25, 28, 29, 35, 36, 38, 39, 44, 45, 47]. An ideal datacenter resource allocator would provide performance isolation between tenants to the extent possible, while gracefully degrading to some notion of fair or differentiated service when resources become congested.

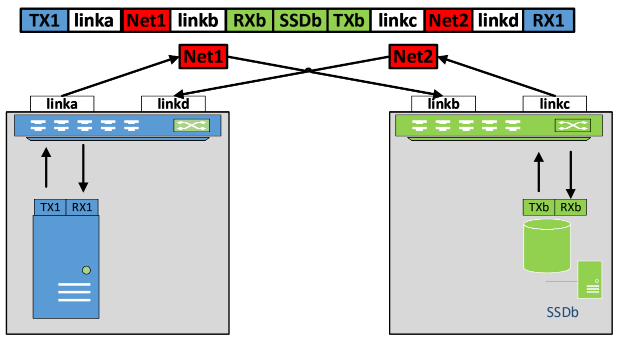


Figure 1 An array (above) of shared resources on path from a VM in a compute rack (left) to a VHD in a storage rack (right).

Overcoming the implications of sharing oversubscribed resources requires the design and implementation of a scalable centralized multi-resource allocator that operates at high frequency due to frequent changes in tenants' VM utilization levels. A centralized resource allocator has the advantage of datacenter-wide visibility that enables better allocation decisions, especially for tenants that span multiple clusters, such as compute, file storage, and database block stores. For instance, Figure 1 illustrates an array of physical resources on the path taken by I/O requests from a tenant with a single VM in a compute rack, to a virtual hard disk in a storage rack, including shared transmit and receive queues at the host hypervisor, a shared storage service, and shared network links at both sides. From time to time any single one of these resources could become a bottleneck---due to sharing with an aggressive tenant---that limits the rate of service obtained by the compliant tenant.

Numerous multi-resource allocation algorithms have been developed and evaluated at modest scale based on a generalization of max-min fairness, known as Dominant Resource Fairness (DRF) [17]. [1] Of these, Extended DRF (EDRF) [32] represents the state-of-the-art due to its precise closed-form expression [5, 10, 13, 16, 20, 30, 31]. DRF considers what fraction of each resource a tenant is demanding, and the resource of which the tenant demands the largest fraction is the tenant's dominant resource. Fairness between tenants is determined according to their respective dominant resources, and at a system level this approach satisfies a number of desirable properties. In particular, DRF allows each tenant to be assigned a share of every resource in the datacenter and guarantees that the tenant will never be worse off than receiving that share, but may be better off if some other tenant is not using their full share. This allows for performance isolation when the shares reflect a lack of over-subscription and a fair resource allocation when they do.

EDRF challenges. Designing a scalable datacenter-scale resource allocator based on EDRF is challenging due to its prohibitive quadratic complexity and the massive scale of today's datacenters.[2]

Because each tenant demands only a fraction of each resource in a small subset of the resources [22], EDRF proceeds in a series of rounds; in each round it finds the most oversubscribed remaining resource and fully allocates it along with proportional amounts of each other resource. As only one resource is fully allocated in each round, the number of rounds required for completion is linear to the number of resources. Furthermore, each round requires consideration of every remaining resource, resulting in a quadratic relationship with the number of resources.

EDRF's complexity is prohibitive when employed at datacenter scale. Today's cloud datacenters consist of around 100K servers each with $O(10)$ cores hence $O(1M)$ resources to be considered. In public clouds, tenants typically employ a handful of VMs (i.e., 80% of tenants deploy 1-5 VMs) [14] and VMs typically utilize just a few cores (i.e., 80% of VMs comprise 1-2 CPU cores) [14]. Under such scenarios, it is very likely that a single oversubscribed cloud datacenter may host 100K-1M tenants. While the quadratic nature of EDRF easily requires minutes to complete at such massive scale, it is often necessary to recalculate allocations at a frequency, or *control interval*, measured in seconds [5], making EDRF impractical at such scales.

Our proposal. We present Deadline Constrained DRF, or DC-DRF, an algorithm designed for performing fair reallocation of shared datacenter resources at cloud datacenter scale in bounded time. The key insight is to trade off a little (bounded) fairness for a significant gain in speed and scalability. DC-DRF includes a control variable, ϵ, which indicates what fraction of a resource we are willing to discard to speed up computation. Doing so, it reduces the number of rounds as more resources are eliminated in each round, dropping the complexity of the algorithm from quadratic to essentially linear (see Lemmas 3.1 and 3.2 for a precise statement). Across successive control intervals, DC-DRF searches for a value of ϵ such that it converges just short of a given deadline.

The improved complexity, however, comes at the cost of error as some resources that would have been allocated by precise EDRF, remain unallocated. We reduce this error via an implementation that is tailored to the underlying hardware, leveraging its parallel nature (cores and vector arithmetic instructions) and utilizing effectively available on-chip cache capacity. The optimized implementation maximizes the number of completed rounds within the deadline, and hence converges to lower values of ϵ and results consequently in lower errors.

The contributions of this paper are:

- We introduce DC-DRF, an adaptive and approximate version of EDRF, whose accuracy and rate of convergence is adjusted by means of a control variable, ϵ. DC-DRF adapts the value of ϵ dynamically across successive control intervals so as to calculate allocations in time bounded by the control interval frequency;
- We provide an efficient implementation of DC-DRF, which is tailored to underlying hardware, so as to significantly reduce the approximation error introduced by ϵ, improving the fairness of allocations estimated by DC-DRF;
- We demonstrate that combining the ideas above enables practical multi-resource allocation at datacenter scale in bounded time, for a wide range

[1] DRF has been deployed in production as one of the resource schedulers in Hadoop, albeit only at cluster scale [1,24].

[2] In contrast, we do not consider the infrastructure for signaling and enforcing multi-resource allocations as a barrier to adoption, having been adequately investigated in the literature [2, 5, 13, 30, 31, 42]

of resource demands, while achieving near-optimal resource allocations and utilization.

We evaluate DC-DRF against EDRF using synthetic inputs modelled on characteristics of a public cloud datacenter [14], where the number of tenants exceeds that of prior work by up to two orders of magnitude and, equally important, there are significant variations in demand between tenants. Our results show that DC-DRF succeeds in enabling multi-resource allocation at public cloud scale, on commodity hardware, in practical time, with much lower error (relative to EDRF) than previous approaches.

2. BACKGROUND AND MOTIVATION

Our interest and motivation for this work dates from 2012 when the Windows Server team challenged us to propose a practical storage QoS architecture for *private* cloud that is capable of preventing performance collapse for tenants sharing physical resources with aggressive tenants. This led to the design of IoFlow [42] which is employed by End-to-End Storage QoS feature of Windows Server 2016 and targets sharing of storage servers in private cloud setups, typically containing hundreds of servers. Extending such architecture to other shared resources led to the design and implementation of Pulsar [5]. Pulsar uses per-resource cost functions and vector rate limiters in hypervisors to a) measure tenants demand, and b) to enforce work conserving reservations according to allocations calculated by a central SDN-like controller running EDRF. While it proved easy to construct effective demand estimation and vector rate limiters, the major challenge lay in the implementation of a centralized resource allocator. In particular, the performance and scalability of EDRF stood out as a fundamental obstacle to scaling Pulsar to *public* cloud datacenter scales.

2.1 Multi-Resource Allocation

Originally introduced for job scheduling in Hadoop clusters, Dominant Resource Fairness (DRF) calculates multi-resource allocations with four properties: (i) sharing incentive---i.e., no tenant would gain from a simple partitioning of resources across tenants; (ii) strategy-proofness---i.e., no tenant can benefit by indicating a false set of resource requirements (demands); (iii) envy-freeness---i.e., no tenant would prefer the allocation made to some other tenant; and (iv) Pareto efficiency---i.e., increasing one tenant's allocation necessarily decreases another tenant's allocation.

DRF computes the share of each resource allocated to each user. The maximum among all shares of a user is called that user's *dominant share*, and the resource corresponding to the dominant share is called the *dominant resource*. DRF simply applies max-min fairness across users' dominant shares---i.e., DRF seeks to maximize the smallest dominant share in the system, then the second smallest, and so on until all resources are exhausted [17].

Input/Output. The algorithm takes as input a *Demand* matrix in which each row represents, for example, a tenant and each column represents a resource. A matrix cell (a_i, r_j) represents the demand of tenant a_i for resource r_j. The output is an *Allocation* matrix whose cells contain an allocation of resources to tenants that satisfies the fairness properties of the algorithm. Optionally, the algorithm supports weights w_{ir} which allow differentiated guarantees.

Algorithm. Algorithm 1 presents the state-of-the-art specification of DRF from [32]. The set of all tenants and resources are denoted by N and R, respectively. Set S_t denotes the set of active tenants at round t. The algorithm takes as input the normalized *demand* matrix, which is calculated by dividing the *Demand* matrix by the max demand resource. The algorithm reports the allocated resources in the *Allocation* matrix, denoted by A, where the proportions between each pair of resources in the demand vector of a tenant are preserved in the allocation vector calculated for that tenant. We adopt the notation from EDRF, summarized in Table 1.

ALGORITHM 1: EDRF

```
Data: Demands d, weights w
Result: An allocation A
1   $t \leftarrow 1$;
2   $\forall r, s_{r1} \leftarrow 1$;
3   $S_1 \leftarrow N$;
4   while $S_t \neq \emptyset$ do
5       $x_t \leftarrow min_{r \in R} \frac{s_{rt}}{\sum_{i \in S_t} \rho_i \cdot d_{ir}}$;
6       $\forall i \in S_t, r \in R, A_{ir,t} \leftarrow x_t \cdot \rho_i \cdot d_{ir}$;
7       $\forall i \in N \setminus S_t, r \in R, A_{ir,t} \leftarrow 0$;
8       $\forall r \in R, s_{r,t+1} \leftarrow s_{rt} - \sum_{i \in S_t} A_{ir,t}$;
9       $t \leftarrow t + 1$;
10      $S_t \leftarrow \{i \in N : \forall r \in R, d_{ir} > 0 \Rightarrow s_{rt} > 0\}$
11  $\forall i \in N, r \in R, A_{ir} \leftarrow \sum_{k=1}^{t-1} A_{ir,k}$
```

Table 1: EDRF Notation

N	the set of tenants (called agents in DRF and EDRF)
R	the set of resources, $r \in R$
D_{ir}	demand of tenant *i* for *r* as a fraction of *r*
d_{ir}	$D_{ir}/(\max_{r'} D_{ir'})$
d_i	$< d_{i1}, \ldots, d_{im} >$ normalized demand vector of *i*
w_{ir}	weight of tenant *i* with respect to resource *r*
A_{ir}	fraction of *r* allocated by DRF to tenant *i*
r_i^*	weighted dominant resource of tenant *i*
ρ_i	$w_{ir_i^*}/d_{ir_i^*}$, weight ratio at dominant resource
s_{rt}	residual fraction of *r* after round *t*
$A_{ir,t}$	fraction of *r* allocated to *i* in round *t*

Intuitively, EDRF calculates the allocation iteratively; each iteration is akin to a round of a *Max-Min* water-filling algorithm. For each resource r, the algorithm maintains the fraction remaining at the start of each iteration, and calculates the fraction that can be added before the water fills a resource. At the end of each round, at least one resource and all tenants who demand it are eliminated---i.e., the resource that determined the quantity allocated in that round.

The complexity of the algorithm is $O(|R|^2|N|)$ as in the worst case there is one round per resource and each iteration of lines 5-8 requires actions per each (tenant, resource) pair. Even with a bound B on the number of demanded resources, complexity is $O(|R||N|B)$. As we show in Section 2.3, the algorithm suffers from scalability issues when the number of tenants and/or resources is large---e.g., 100s of thousands.

2.2 Short Control Intervals

In datacenters with high workload variation, it is important that resource allocations are recalculated at small control intervals so as to reduce the negative impact of staleness of resource demands. Short control intervals allow for accommodating the resource requirements of new tenants and idle tenants who become active, as well as reclaiming resources from tenants who become less active. Being reactive to the arrival of new tenants is particularly important. Without prior demands, these tenants have no allocations for resources; hence, we can either delay their progress until the next control interval, or allow them to ramp up unchecked until the next control interval. In the former case, a short control interval benefits tenants who are ramping up by allowing them to do so sooner, and in the latter case it benefits active tenants sharing resources that become temporarily overloaded by tenants ramping up---i.e., the so-called *noisy neighbor* effect.

We substantiate the need of short control intervals by studying the variation in CPU load utilization of Azure virtual machines (VMs) [14]. Figure 2 plots the number of VMs that transition at least once within a five-minute interval either from an idle state to an active one or from an active state to an idle one. A VM is considered to become active within an interval in case that either its maximum CPU utilization is at least 3x higher than its average CPU utilization, or its maximum CPU utilization is at least 10% (absolute value) higher than its average CPU utilization (e.g., 20% to 30%). Likewise, a VM with non-negligible CPU utilization (at least 5\%) is considered to become idle within an interval in case that either its minimum CPU utilization is at least 3x lower than its average CPU utilization, or its minimum CPU utilization is at least 10% (absolute) lower than its average CPU utilization. The measured 95th percentile of the fraction of VMs that become active (29%) and idle (13\%) over a period of 350 five-minute intervals corroborates earlier findings that VMs exhibit bursty traffic patterns [14].

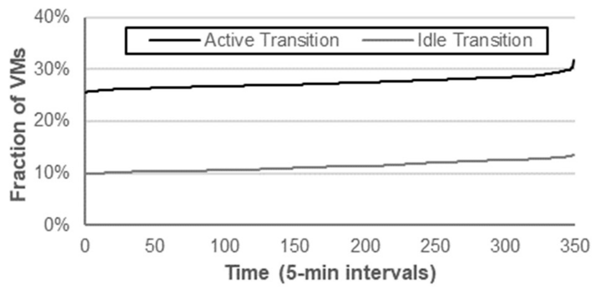
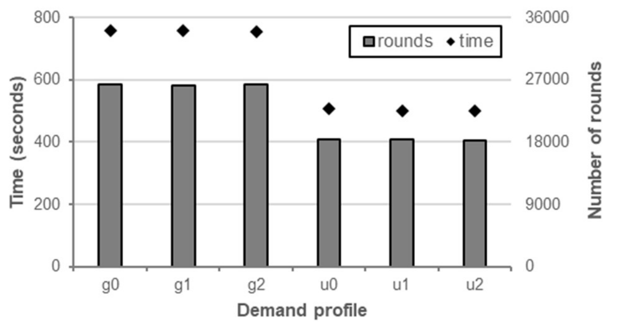


Figure 2 Fraction of VMs that become active or idle at least once within each 5-min interval.

Prior work in resource management advocates for control intervals ranging from 10 seconds to 30 seconds due to high workload variation in enterprise servers [5]. In order to obtain conservative estimates of datacenter workload variation at the proposed control intervals, we assume that the observed transitions: (a) occur only once within each five-minute interval and (b) are spread uniformly across all control intervals. For the proposed (10-second -- 30-second) intervals, 1.4-4.2% of VMs exhibit at least one state transition, indicating significant churn in the level of activity for a significant fraction of VMs. This VM-based estimate is low because we care about fairness on the level of workloads which span multiple VMs. This finding motivates the choice of a short control interval, such as ten seconds.

2.3 Scalability Limitations

We now demonstrate that conventional DRF is impractical at datacenter scale using a hypothetical datacenter modelled on modern cloud datacenters. Modern datacenters consist of around 100K servers arranged in racks of 20-40 servers each, connected to a shared multi-tier CLOS network built from 40-100Gbps Ethernet [22]. Persistent data reside in shared *storage* racks, and customer VMs are packed into *compute* racks, so that VMs of each tenant access data and each other over a shared multi-resource environment. Access patterns exhibit variable degrees of locality [9, 11, 26, 34, 39]. Tenants are predominantly small, nearly 80% of VMs have only 1 or 2 cores and 80% of tenants have 5 or fewer VMs [14]. Thus, with multiple cores per server, the number of tenants, or *tenants*, may easily exceed the number of servers.



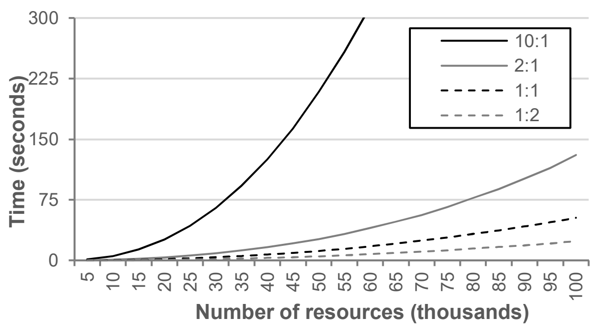


Figure 3 Conventional DRF performance analysis: (Left) Number of rounds and time for 100K resources and 1M tenants. Key: demand vector lengths [0,128] chosen by Gaussian (g) or Uniform (u), resource locality is none (0), single cluster (1) or two clusters (2); (Right) Scalability under variable scale and different Tenant:Resource ratios.

We generate synthetic multi-resource workloads for a 100K server datacenter supporting 1M tenants (Section 5.1). Figure 3 (left) plots the number of rounds and elapsed time using a basic implementation of DRF such as found in FILO [31] and Yarn [1]. Figure 3 (right) plots runtime as the number of resources increases for several (tenant:resource) ratios, demonstrating a non-linear relationship between completion time and scale. The number of rounds highly depends on the size (or scale) and diversity of the demand matrix. DRF may complete in a small number of rounds for either (a) small or (b) highly symmetric demand matrices. Cases where this is approximately true have been exploited by prior work [13]. In contrast, large inputs with high diversity require large execution times.

2.4 Suboptimal Solutions

Resource aggregation. We have investigated hierarchic structure and aggregation as a possible abstraction for limiting the complexity of DRF, and found that aggregating resources and/or tenants can introduce error and unfairness, as follows. Consider two tenants, A and B, with demand vectors ranging over the sets $\{r_1, r_2\}$ and $\{r_2, r_3\}$, respectively. In an attempt to decrease the number of tenants in the input to DRF, we construct an aggregate demand vector from the union of the above sets and sum the demand for any resources that appear in both sets, in this case r_2. Recall from Section 2.1 that DRF evicts a tenant in the round where one (or more) of its resources is saturated, thereby preserving demand proportions. If r_1 is the only resource saturated in that round, then B will be denied its share of the residual capacity remaining at r_2 and r_3 (likewise for A and r_3.) This finding compares to H-DRF in that while H-DRF does have a hierarchical structure, it does not rely on aggregation ensuring fairness but without improvement in complexity [10].

Early termination. Terminating EDRF after a reduced number of rounds (e.g., one round) introduces error. While it is guaranteed that the most demanded resources are fully allocated at the end of a round, the remaining resources are allocated proportionally. Depending on the ratio of resources allocated to resources not allocated, early termination may result in severe resource under-utilization and unfairness. Section 5 discusses and quantifies the implications of early termination on resource utilization and fairness.

3. DC-DRF

Algorithm 2 gives the specification for DC-DRF, which comprises an outer loop and an inner loop. Each iteration of the inner loop performs multi-resource allocation using an approximation of DRF whose time and relative error are determined by a control variable ε. The outer loop represents consecutive control intervals of duration Δ_α and has a deadline τ after which the inner loop will terminate on completing its current iteration. Over consecutive control intervals the outer loop searches for a value of ε such that the inner loop completes just within the deadline. The inner loop of DC-DRF derives from EDRF and our description highlights two key algorithmic changes, that are key in reducing the number of iterations and execution time of each iteration, respectively.

Element ϵ. DC-DRF introduces the approximation variable $\epsilon \ll 1$, which relaxes the decision of when a resource is considered to be exhausted. Most importantly, this variable provides a mechanism for constraining the number of iterations required for the algorithm to converge and terminate.

DC-DRF considers a resource $r \epsilon R$, to be exhausted when the residual capacity of r falls below the value of ϵ, thus causing early exhaustion of resources and early eviction of tenants compared to DRF. The early eviction implements a form of approximation, achieving faster termination than DRF, but introducing error relative to the allocations obtained by DRF.

ALGORITHM 2: DC-DRF

```
1   $j = 1$;
2   $\epsilon_1 \leftarrow 0$;
3   **while true do** /* once per control interval */
4     $\Delta_j \leftarrow time() + \Delta_\alpha$;
5     $t \leftarrow 1$;
6     $\forall r, s_{r1} \leftarrow 1$;
7     $S_1 \leftarrow N$;
8     $\mu_{1r} \leftarrow \sum_{i \in S_t} \rho_i \cdot d_{ir}$; /* see Element $\mu$ */
9     $y_0 \leftarrow 0$
10    $z_i \leftarrow 0$
11    $\tau_j \leftarrow false$;
12    **while** $S_t \neq \emptyset$ **and not** $\tau_j$ **do** /* inner loop */
13      $x_t \leftarrow \min_{r \in R} \frac{s_{rt}}{\mu_{tr}}$
14      $y_t \leftarrow y_{t-1} + x_t$
15      $\forall r \in R,\ s_{r,t+1} \leftarrow s_{r,t} - x_t \mu_{tr}$
16      $S_{t+1} \leftarrow \{i \in N: \forall r \in R, d_{ir} > 0 \Rightarrow s_{r(t+1)} > \epsilon_j\}$
17      $\forall r \in R, \mu_{t+1,r} \leftarrow \mu_{t,r} - \sum_{i \in S_t \setminus S_{t+1}} \rho_i \cdot d_{ir}$
18      $\forall i \in S_t \setminus S_{t+1},\ z_i \leftarrow y_t$
19      $\tau_j \leftarrow time() > \Delta_j$
20      ***if*** $\boldsymbol{\tau_j}$ ***then*** /* timeout occurred */
21        $\forall i \in S_{t+1},\ z_i \leftarrow y_t$
22      $t \leftarrow t + 1$
23    $\forall i \in N, r \in R, A_{ir} \leftarrow \rho_i \cdot d_{ir} \cdot z_i$ /* see *y*, *z* */
24    ***if*** $\boldsymbol{\tau_j}$ ***then***
25      $\epsilon_{j+1} \leftarrow raise(\epsilon_j)$
26    ***else***
27      $\epsilon_{j+1} \leftarrow lower(\epsilon_j)$
```

Observe in line 16 that DC-DRF replaces the constant zero of EDRF line 10 with the control variable ϵ_j. In the event that ϵ_j has value zero then the two constructs are equivalent; however, non-zero ϵ_j causes early exhaustion of the resource and early eviction of any tenant that has non-zero demand for that resource. The inner loop is terminated if its elapsed time exceeds the deadline Δ_j with timeout indicated in τ_j (lines 11, 19, 20, 24).

The outer loop, lines 3-11 and 23-27, represents consecutive control intervals, j, over which DC-DRF searches for a value of ϵ such that the inner loop can complete just ahead of a time deadline Δ_α. We assume that inputs vary from one interval to the next and that DC-DRF must continually adjust ϵ to adjust for the variations in execution time. In the event of timeout the value of ϵ is increased, otherwise it is decreased, by the function *raise()* and *lower()*, respectively. We do not specify the precise search strategy followed by *raise()* and *lower()*; in our implementation we maintain a search window with initial fast-start gradient. The intention is, given stable inputs, DC-DRF will oscillate between an ϵ that completes just within the deadline, and timeout that tests a slightly lower ϵ.

Element $\boldsymbol{\mu}$. EDRF evaluates the expression $\sum \rho_i \cdot d_{ir}$ in each iteration of its body at line 5. This entails repeatedly summing elements whose value has not changed from one iteration to the next. Based on this observation, DC-DRF introduces the element μ into which it performs the summation prior to entering the iterative part of its body (line 8). Subsequently, DC-DRF subtracts from μ the values associated with evicted tenants once at the time of the tenant's eviction (line 17). This optimization allows us to compute the values of x_t for each t while performing 2 operations per tenant-resource pair rather than 1 operation per tenant-resource pair per inner loop iteration. Relatedly, note that ρ_i and d_{ir} only appear as the product $\rho_i \cdot d_{ir}$, and thus this product can be computed a single time per outer loop iteration. (For ease of comparison to EDRF, we omit this from the description in Algorithm 2.)

Elements $\boldsymbol{y}$ and $\boldsymbol{z}$. EDRF calculates intermediate values of $A_{ir,t}$ at line 6. We observe that computing all of these values $x_t \cdot \rho_i \cdot d_{ir}$ requires several operations per tenant-resource pair per round. Instead we can keep these implicit using y and z, allowing them to be computed just once per tenant-resource pair. We use y_t as an accumulator to hold a running total of the sum of all x_t (line 14), and use z_i to cache for each tenant the value of y_t at the time of the tenant was evicted (typically line 18, or line 21 when terminating on τ_j). In this way $A_{ir.t}$ is calculated once per tenant on exit from the inner loop (line 23). In addition, factoring things this way allows the sharing of some additional computations across tenants (we compute one y_t per round rather than needing to do this addition for each tenant). Not only does this decrease the total number of arithmetic operations performed, it also improves the spatial locality of the algorithm by eliminating references to A_{ir} from the inner loop: this is important for performance and therefore for precision, as later sections will show.

Computational Complexity. The variable ϵ is crucial for both the computational complexity of the algorithm. The key feature for the time complexity of the algorithm is the number of iterations of the inner loop required. For EDRF, there will be $O(|R|)$ iterations, as barring ties in line 5 only one resource is exhausted per iteration. For a simplified version of DC-DRF, we can show that the number of iterations is independent of $|R|$. In particular, suppose that μ_{tr} is not updated in the course of the inner loop and instead is always fixed at μ_{1r}. Furthermore, suppose that there is a constant c such that $\min r, r' \in R \frac{\mu_{1r}}{\mu_{1r'}} > c$. That is, there is a bound on how large the relative demands for two resources are.

Lemma 3.1 *Under the two assumptions the number of iterations of the inner loop is* $O(log^{-1}(1-\epsilon))$, *independent of* $|R|$.

Proof. The first iteration completes with y_1 sufficient to exhaust some resource $r_1(y_1\mu_{r_1 1} = s_{1r_1} = 1)$). Any resource r still demanded by some tenant in S_2 must have at least an ϵ fraction remaining ($y_1\mu_{r1} < 1 - \epsilon$). Taking advantage of our first assumption, we can express the more general version of this fact for each iteration as $y_t\mu_{r_t 1} = 1$ for the resource r_t exhausted in iteration t and $y_t\mu_{r1} < 1 - \epsilon$ for any resource r still available in iteration $t+1$. In particular, this holds for r_{t+1}. This means that

$(1-\epsilon)\mu_{r_t 1} > \mu_{r_{t+1}1}$ for all t. These inequalities telescope to give $(1-\epsilon)^{t-1}\mu_{r_1 1} > \mu_{r_t 1}$. By our second assumption, this puts a bound on the number of iterations that $(1-\epsilon)^{t-1} > c$. Solving for t gives the desired bound. ■

The proof of the lemma suggests how pathological inputs can be constructed to cause DC-DRF to require $|R|$ iterations: cause the μ_{tr} to decay in such a way that there is always only one resource exhausted per iteration. Note however that, (i) such inputs require a coordination of demand which seems unrealistic in a real system, (ii) in such cases DC-DRF effectively falls back to the performance of EDRF, and most importantly (iii) whether a given example is pathological depends on ϵ, so we would expect our outer loop to drive ϵ smaller and restore good performance. Thus we believe such inputs have no relevance in practice.

We can now quantify the overall computational complexity of an iteration of the outer loop.

Lemma 3.2 *The computational complexity of an iteration of DC-DRF is* $O(|R||N|)$. *Furthermore, let B be a bound on the number of resources a tenant demands and let I be the number of iterations of the inner loop. Then the computational complexity of an iteration of DC-DRF is* $O(|N|B + I|N| + I|R|)$

Proof. Non-trivial amounts of computation occur in lines 8, 13, 15--18, 21, and 23. Per the discussion of μ, the combined complexity of lines 8 and 17 is $O(|N|B)$. For lines 13 and 15 is $O(I|R|)$, while for line 16 it is $O(I|N|)$. By hoisting the computation of allocations into the outer loop, lines 18, 21, and 23 have combined complexity $O(|N|B)$. Since $I \leq \min(|N|, |R|)$ and $B \leq |R|$, the result follows. ■

Lemma 3.2 shows that our more careful implementation reduces the complexity of EDRF from $O(|R|^2|N|)$ to $O(|R||N|)$ for the case without a bound B, and in this case ϵ does not improve the overall complexity, which is dominated by calculating the initial demand for resources (line 8). However, with such a constant bound B, under the assumptions of Lemma 3.1 the gain from using ϵ is substantial (as I is now constant for fixed ϵ, from $O(|R||N|)$ to $O(|N|)$. If $|R|$ and $|N|$ are of similar magnitude, as we expect in public cloud settings, this is a reduction from quadratic to linear.

Approximation Quality. The variable ϵ is crucial for the quality of the approximation, but its effects of ϵ are somewhat subtle. The simplest concrete statement is that if the inner loop runs to completion (i.e. there is no timeout) then every tenant desires some resource of which we have allocated a $1-\epsilon$ fraction. However, this does not imply a $1-\epsilon$ fraction of each tenant's optimal utilization because different tenants will have different bottleneck resources (although our results in Section 5 show that it does empirically). Additionally, consider the scenario in which ϵ causes a tenant to get less than is "fair." If all tenants using the tenant's bottleneck resource desired more, the tenant could get at most $\frac{1}{1-\epsilon} \approx 1+\epsilon$ times its actual allocation, which is an insignificant difference. The possibility to have large gains requires other tenants be unable to use more because they are bottlenecked on some other resource. Thus the "unfairness"' corresponds to not getting an outsize share of a heavily loaded resource when others rely on an even more heavily loaded resource. Given the benefits of providing fairer and more efficient resource allocation overall, this particular ``unfairness" may well be worth tolerating.

Finally, EDRF was motivated by four desiderata [17]. Despite our approximation, DC-DRF fully satisfies two of them (i.e., sharing incentive and envy-freeness), while partially satisfying Pareto efficiency and strategy-proofness up to the resources discarded by ϵ.

4. DESIGN AND IMPLEMENTATION

4.1 High-Level Design

The scope of our work presented herein is limited to the design and implementation of the DC-DRF algorithm itself. While we have not attempted to integrate it into datacenter infrastructure, we provide context for the reader we sketch a design of how we envisage that might work.

In outline, DC-DRF would provide a centralized service for calculating multi-resource allocations in a public cloud datacenter of around 1M resources, such as individual servers or storage volumes, and one million tenants. Separate admission control and VM placement services would provide DC-DRF with initialization data including resource identities. Trusted components within infrastructure report demands and enforce allocation limits; Pulsar demonstrated the use of cost functions in trusted hypervisor drivers to generate dynamic demand vectors based on the queued and in-flight operations of each tenant [5]. The frequency, or control interval, with which allocations are revised needs to be in the order of seconds, so as to rapidly curtail excessive resource consumption by tenants whose demand suddenly becomes excessive, and to reclaim resource allocations from tenants whose demand has dropped.

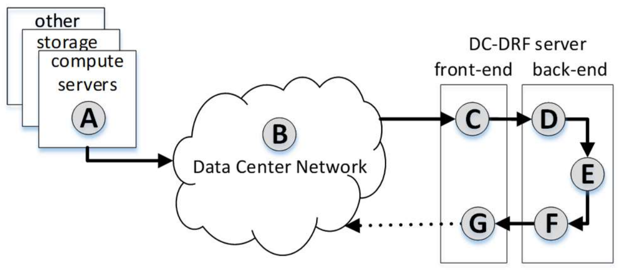


Figure 4 A central DC-DRF server receives tenant demands from trusted infrastructure components and returns allocations for enforcement.

Referring to Figure 4, infrastructure components (A) determine a demand vector for each tenant by sampling queue lengths, sending these over the datacenter network (B) to a front-end server (C). The front-end offloads communication overheads from the performance-sensitive back-end, which runs on a dedicated server. The front-end loads tenant demands into the input buffer (D) of the back-end DC-DRF process (E), where allocations for the current control interval are calculated and loaded into an output buffer (F). The front-end (G) retrieves the allocations and sends them for enforcement in the infrastructure (A). The network overheads are less than 2 Gbps.[3]

4.2 Evaluation Platforms

Unless stated otherwise, we use a dual-socket Dell R740 server with Intel Xeon Platinum 8160 CPUs. Each CPU employs 24 cores operating at 2.1GHz and a shared 33MB L3 cache.[4]

4.3 Data Structures

In order to avoid memory allocation overheads at runtime, data structures are statically allocated during initialization to maximum sizes given by a runtime parameter. The first thread to run allocates the central data structures and initializes a set of worker threads, each of which allocates the private data structures and scratch space for its own use.

Each tenant has a data structure encoding its demand, weight and accumulated allocation vectors for each resource for which it has non-zero demand. DC-DRF implements a sparse encoding of demand and allocation matrices, imposing an upper bound B on the number of resources that a tenant may demand. By default, B=128. This optimization helps reduce the memory footprint and working set of our implementation, alleviating pressure on CPU caches and memory bandwidth.

4.4 Arithmetic Precision

Assuming a worst case of 1M tenants each demanding an equal share of a given resource, then each tenant would receive a $1/1M$ allocation and this value sits comfortably within the range $[5.96 \times 10^{-8}, 65504]$ of IEEE 754-2008 half-precision (16-bit) floating point. In contrast, 16-bit integer arithmetic underflows to produce $\frac{1}{N} = 0$ for $N \geq 65536$. At public cloud datacenter scale the number of tenants sharing a resource could generate denominators much larger than that.

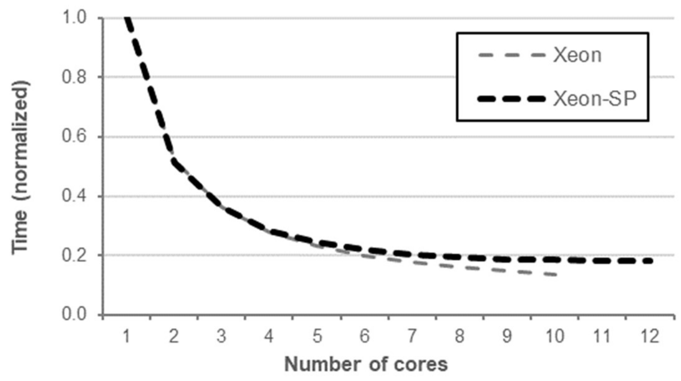
Contemporary processors do not support 16-bit floating point; hence, we choose between 32-bit (single-precision) and 64-bit (double-precision) at compile-time. Figure 6 shows that single-precision consistently outperforms double-precision, primarily due to better Last Level Cache (LLC) performance. The improved LLC performance is attributed to higher temporal and spatial locality. First, the working set of single-precision setups is half of double-precision, thereby alleviating cache pressure at high thread count. Second, as caches utilize a 64-byte cache block, more single-precision variables are fetched upon a cache miss, leading to higher cache hit rates.

4.5 Thread-level Parallelism

We achieve parallelism by tiling the data space so that independent threads can proceed on distinct partitions, synchronizing when necessary on custom lock-free barriers based on atomic counters and busy-waiting. We employ two types of tile, namely *tenant tiles* and *resource tiles*. Tenant tiles partition the set of tenants, and resource tiles partition the set of resources. Each tile is assigned to a distinct thread. Careful memory alignment of resource tiles along cache-line boundaries avoids false sharing: the worker threads do not collide when updating fields within shared data structures. The two types of tile can be imagined as horizontal and vertical stripes over a 2D matrix. To illustrate, consider a system comprising *n* NUMA nodes each with *c* cores. Table 2 summarizes how tiling is applied within the inner loop, and the level of parallelism (MPL) achieved.

[3] Assume each tenant has demand for 128 resources and encodes this as a list of (resource,demand) integer pairs. With 1M tenants the DC-DRF server receives $1M \times 128 \times 4 \times 4$ bytes of demand data per control interval, and sends the same in allocation data.

[4] We have also tested on a Dell R730 dual-socket server with Intel Xeon E5-2660v3 processors of 10 cores each 2.6GHz and a shared 25MB L3 cache, typical of legacy servers still found in Public Cloud datacenters today.



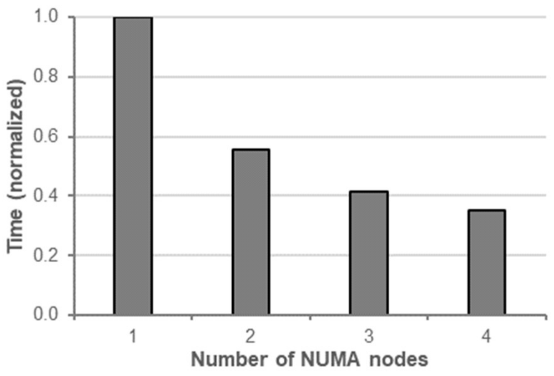


Figure 5 Multi-threaded scalability: (Left) Core-level scalability with hyperthreading enabled (two threads per core); (Right) NUMA-level scalability.

Table 2: Degree of parallelism (MPL) for *n* NUMA nodes of *c* cores each.

Line	Tiles	MPL (min)	MPL (max)
13	Resource	1	c*n
15	Resource	c*n	c*n
16	Tenant	c*n	c*n
17	Resource	c	c*n

Figure 5 (left) shows the elapsed time of a DC-DRF microbenchmark as we vary the number of cores, using hyperthreading and 32-bit variables throughout. Each step adds two threads to fully load each core (with hyperthreading the second thread on each core invariably provides less benefit than the first thread). With a small number of cores the computation is CPU-bound, and increasing the number of cores leads to a clear decrease in elapsed time. While adding more cores improves performance, the scalability is sub-linear to the number of cores and gradually the benefit of adding more cores decreases because (a) increased inter-thread contention for LLC space; as our data structures greatly exceed the LLC size, this contention spills over into demand for memory bandwidth for moving data between CPU caches and DRAM and (b) the existence of variables $s_{r,t}$ and $\mu_{t,r}$ shared across all threads require the use of synchronization primitives, thereby introducing a non-negligible sequential part which becomes dominant as the number of threads increase and the parallel part becomes smaller.[5]

4.6 NUMA Awareness

To eliminate unnecessary cross-NUMA memory accesses over QPI [3], we explicitly allocate memory to each thread from the DRAM attached to that thread's NUMA node, and perform intermediate aggregation of partial results at each NUMA node prior to calculating global results for each round of DRF.[6]

We measure the benefit obtained from additional NUMA nodes by running a micro-benchmark on a Dell R910 server featuring four NUMA nodes, increasing the number of NUMA nodes at each step. The results are shown in Figure 5 (right). Because each NUMA node employs its own LLC, adding a second NUMA node improves performance by almost 50 percent. However, the benefit from additional NUMA nodes decreases because, while the cache bottleneck is alleviated by the extra LLC capacity provided by each NUMA node, the sequential phase in each round of the algorithm limits the total performance.

4.7 Vector Arithmetic

Contemporary processors support 256-bit vector instructions (AVX-256) [4] while recently announced Intel Xeon-SP adds support for 512-bit vectors (AVX-512) capable of operating on eight double-precision or 16 single-precision values. The latter also adds a scatter instruction for moving values from a 512-bit vector registers into memory, an attractive feature given the sparse random memory access patterns of DC-DRF. Our implementation is optimized for both instruction sets.

[5] We choose C++ memory barriers over the native SYNCHONIZATIONBARRIER of our host OS as, while the latter provided correctness, its performance over multiple NUMA nodes was poor, leading to a drop-off in performance when operating over more than one NUMA node.

[6] Our findings indicate the default OS allocator fails to choose local DRAM---i.e., DRAM that is connected to the NUMA node to which the invoking thread was bound.

ALGORITHM 3: Example use of AVX512 vector intrinsic functions from our implementation.

```
1    __mm512_vindex vindex_512 = _MM512_LOAD_VINDEX(*ptr);
2    __m512r mu_tr = _mm512_i32gather_pr(vindex_512,pScratchR);
3    mu_tr = _mm512_add_pr(mu_tr, A_irt);
4    _mm512_mask_i32scatter_pr(pScratchR, m, vindex_512, mu_tr);
```

To illustrate with an example, Algorithm 3 shows AVX512 [4] intrinsic functions from our implementation of DC-DRF line 17. At this point each thread is accumulating allocation vectors A_{ir} from its agent tile into a thread-local per-resource array eventually to be subtracted from $\mu_{t,r}$. Line one loads an AVX register with indices identifying up to 16 resources (8 resources if compiled for 64-bit, throughout) from an agent's allocation vector, A_{ir}, which line 2 uses to gather (fetch from memory) the accumulated allocations made by this thread in this round for the resources identified in line one. Line 3 adds to these accumulated values the vector A_{ir} of the allocation made to agent i in this round. Finally, line 4 scatters (stores to memory) the newly updated values into the array accessed in line 2. Each thread executes this code in parallel, following which the thread-local results are aggregated at NUMA and global level using resource tiles.

As shown in Figure 6, AVX-256 and AVX-512 deliver similar performance. Both AVX-256 and AVX-512 improve performance over the baseline scalar version by 25%. To our surprise, AVX-512 provides marginal performance gains over AVX-256 that are within the statistical error. We shed light on this surprising result through a micro-architectural analysis of all configurations, finding a trade-off between (a) instruction count and memory-level parallelism (MLP) and (b) operating core frequency.

First, vectorization reduces the instruction count by up to 21% and improves MLP (i.e., number of concurrent outstanding long-latency memory accesses [15]) by up to 8% allowing for overlapping memory stalls with computation. The small gain in MLP is attributed to the fact that the core is capable of extracting MLP from scalar loads and stores due to its out-of-order execution and large instruction window; hence, the gather/scatter instructions provide relatively small gains.

Second, cores operate at lower frequency when using AVX-256 and AVX-512, by 3% and 11%, respectively. The observed lower frequency is due to the higher number of switching transistors when vector instructions are executed (due to larger register file and execution unit's datapath). To account for this, core frequency is scaled down so that power/thermal limits are not exceeded. The resulting lower frequency offsets the performance gains of larger degree of vectorization. In the case of AVX-512, we observe diminishing returns as the drop in frequency is modest while instruction-count and MLP are improved over AVX-256 by a small factor.

4.8 Baseline Configuration

Figure 6 shows a parameter sweep of significant configuration options, ranked left to right in order of decreasing completion time. The right-most columns indicate the configurations that complete the same computation in the shortest time. While hyper-threading improves performance of scalar versions, it provides negative performance gains when vectorization is enabled. This is due to the fact that hyperthreads introduce a high degree of destructive inter-core and intra-core sharing: (i) hyperthreads contend against each other for core's vector unit which is utilized even when hyperthreading is disabled and (ii) inter-core contention for shared caches; thus making hyperthreading ineffective for parallel vectorized sections at high thread counts. Based on our findings, we use 32-bit floating point and 512-bit vector instructions without hyperthreading throughout Section 5.

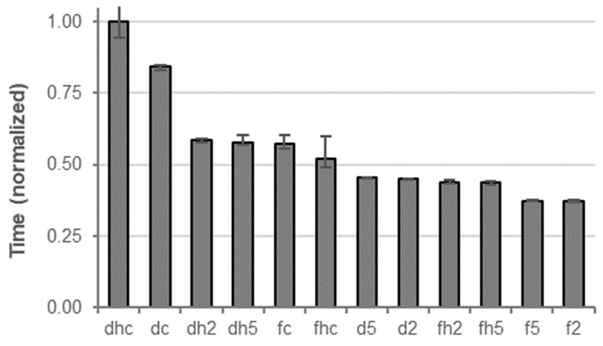


Figure 6 Configuration options ranked by runtime. Key: double (d) or single (f) precision, scalar (c), AVX256 (2), AVX512 (5), hyper-threading (h).

5. EVALUATION

We now report the results of our experiments to explore the precision of allocations calculated by DC-DRF, and its performance limits running on commodity servers.

5.1 Methodology

Demand profiles. We evaluate DC-DRF on datacenter-scale inputs using demand profiles whose parameters model the workload characteristics of public cloud datacenters [14, 22]. Table 3 summarizes the different demand profiles we use throughout the evaluation.

Table 3: Demand profiles: U=Uniform, G=Gaussian, Resource locality: DC=datacenter, PodA=1st cluster, PodB=2nd cluster

key	vector len	select resource	res.
U0	U[2,128]	U(DC)	U(R)
U1	U[2,128]	U(PodA*.5,PodB*.5)	U(R)
U2	U[2,128]	U(PodA*.5,PodB*.3,DC*.2)	U(R)
G0	G[2,128]	U(DC)	U(R)
G1	G[2,128]	U(PodA*.5,PodB*.5)	U(R)
G2	G[2,128]	U(PodA*.5,PodB*.3,DC*.2)	U(R)

Number of elements. We fix the number of resources at one million and the number of tenants at one million throughout [22].

Demand vector size. Cortez et al. recently contributed the first characterization of a full-scale Public Cloud VM workloads, finding that they are dominated by small tenants: 80% of tenants use only 1-5 VMs and 40% of tenant VMs have a single virtual core [14]. We model this using a truncated Gaussian distribution to generate demand vectors drawn from the interval [2,128]. We also use a uniform distribution that better represents the substantial first-party (cloud provider) workloads.

Resource selection. Traffic between VMs collocated at rack level exhibit strong locality [9, 26]. In contrast, Internet services, such as search and social media, exhibit similar locality only for *heavy-hitter* flows at rack level [37]. We model three different degrees of locality: first, uniform selection across the entire datacenter; second, 50% of resources selected locally within a single cluster of the datacenter; third, 50% of resources selected within one cluster and 30% within a second cluster.

Demand vector values. The DC-DRF and DRF algorithms are sensitive to the value of each element in a demand vector. To minimize correlation between the demand at each resource across tenants, which could artificially accelerate the rate at which resources are exhausted, we select uniformly in the interval [1,C[r]] where C is the absolute capacity of resource r.

DC-DRF configurations. We evaluate four run-time configurations: a) parallel DC-DRF (*DC-DRF*); b) single-threaded DC-DRF (*sDC-DRF*); c) parallel EDRF terminated at the deadline (*pEDRF*); d) parallel EDRF allowing just one round of the inner loop akin to HUG [13] (*pEDRF-1*). pEDRF isolates the benefit of parallelism, and pEDRF-1 highlights the fundamentally iterative nature of EDRF. We report results normalized to a baseline of conventional single-threaded EDRF which completes in 129 minutes.

5.2 Sensitivity to Demand Profiles

We use range of diverse demands to show that DC-DRF has not been inadvertently specialised to narrow inputs that cannot be guaranteed in practice, such as choosing highly symmetric demands [13]. We set the deadline at 8 seconds, based on results of Section 5.5. Figure 7 shows the number of rounds for each demand profile normalized to those of the baseline. In these cases the use of approximation in DC-DRF has decreased the number of rounds by an order of magnitude because more tenants are evicted in each round. Following FILO [31], we measure fairness using the standard deviation between the allocation obtained by DC-DRF and the baseline allocation as shown in Figure 7. The low standard deviation from baseline, which remains very low throughout, indicates that there is no significant compromise made to fairness because the allocations obtained by DC-DRF lie close to those of EDRF which we take as ground truth.

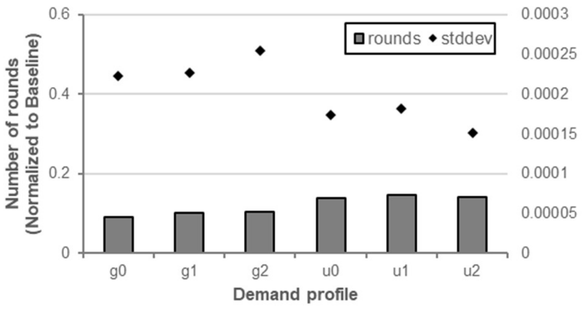


Figure 7 DC-DRF performance and fairness by demand profile. Bars show rounds normalized to baseline. Markers show standard deviation against baseline. Key: demand vector length [0,128], Gaussian (g), Uniform (u), resource locality none (0), one (1) or two (2) clusters.

5.3 Resource Utilization

Henceforth, unless stated otherwise, we use the G0 demand profile, vary the deadline in steps of one second and measure overall resource utilization summed over all resources. Figure 8 plots resource utilization normalized to baseline. Both sDC-DRF and DC-DRF achieve near-optimal utilization for deadlines to the right of the knee in Figure 8 (discussed in in Section 5.5).

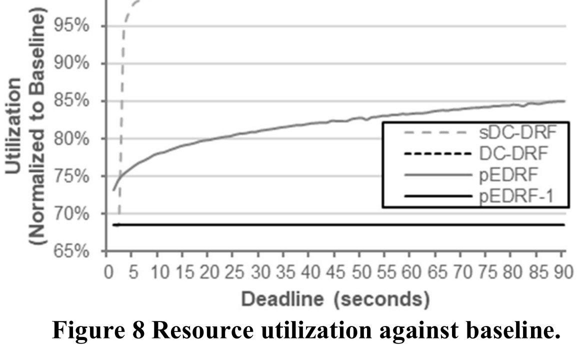
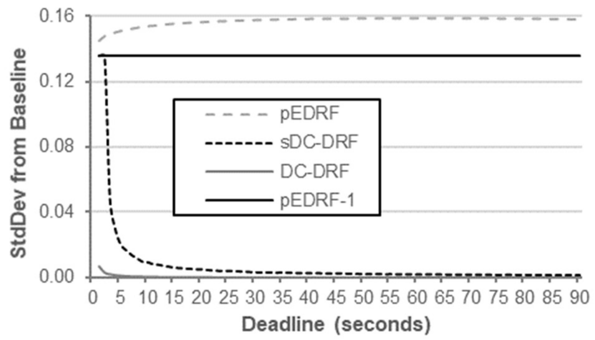


Figure 8 Resource utilization against baseline.



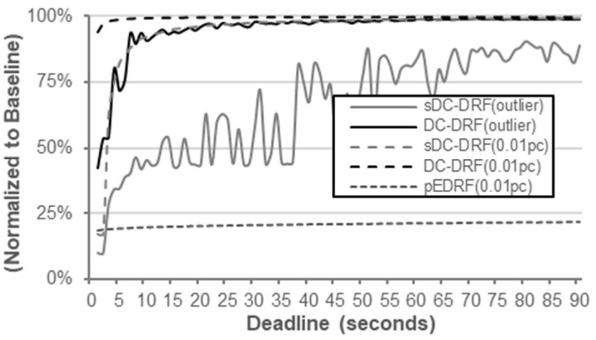


Figure 9 Fairness analysis of various DRF configurations: (Left) StdDev of experimental allocation from baseline allocation; (Right) Under-allocation of outliers compared to baseline. Outlier shown as solid lines, 0.01 percentile as dotted.

In general, outside of narrow corner-cases, pEDRF-1 performs poorly because only those resources selected by EDRF line 5 will be fully allocated in the first round; this set may consist of only one resource while the remaining resources will be allocated proportionally at each tenant. pEDRF enables higher resource utilization as more iterations are completed. pEDRF achieves a utilization that converges with baseline utilization as deadlines approach its completion time.

5.4 Fairness Analysis

Figure 9 (left) shows the standard deviation from baseline for increasing values of deadline. The knee in the curves of sDC-DRF and DC-DRF is explained in Section 5.5, and when DC-DRF is operating to the right of the knee the stddev remains small, indicating a high degree of fairness for epsilon values in that range.[7] The standard deviation for pEDRF starts high but converges to zero as the deadline approaches the time within which it can complete. The standard deviation for pEDRF-1 is high throughout, making it a poor choice of configuration. Note that the error for pEDRF exceeds that of pEDRF-1 due to a higher variance between individual tenants, where instead the variance of pEDRF-1 remains uniformly poor throughout.

However, the standard deviation may conceal outliers in the form of tenants whose deviation from baseline lies far from the mean. Of primary concern are tenants whose allocation under DC-DRF falls below their baseline allocation, because for them DC-DRF is in some sense less fair. To expose the fraction of tenants effected in this way we take the results for the DC-DRF configuration, rank them on deviation from baseline allocation, and show in Figure 9 (right) the under allocation for the tenant with the greatest shortfall, together with the shortfall at the first tenth of a percentile (i.e., the 1000 tenants with the greatest shortfall). The shortfall for DC-DRF is substantially better than that of sDC-DRF, for both the outlier and at one tenth of a percentile.

5.5 Value of Epsilon

We have repeatedly observed a distinct knee in the results of preceding sub-sections, which we now explain. Figure 10 plots the values of epsilon to which DC-DRF converges for the sDC-DRF and DC-DRF configurations. Both configurations feature an initial steep gradient for short deadlines, with a knee leading to a plateau after a few seconds. To the left of the knee, co-variant with the curves seen in earlier experiments, lies a region where DC-DRF is forced to terminate within a very small number of rounds, requiring aggressive values of epsilon. In particular, the first round considers the entire tenant and resource sets independent of the value of epsilon. Note that the values for DC-DRF, shown on the left hand axis, are two orders of magnitude lower than those of sDC-DRF, shown on the right-hand axis, reaffirming that DC-DRF achieves better fairness than sDC-DRF.

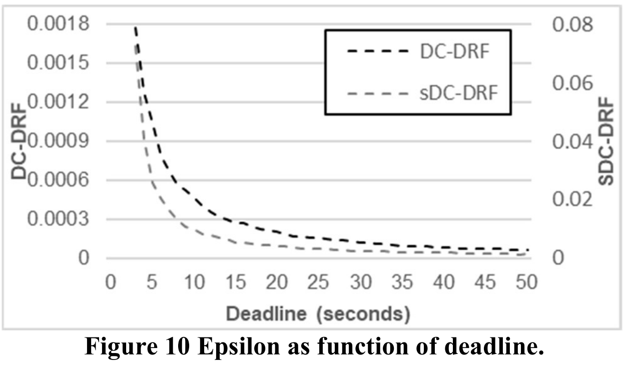


Figure 10 Epsilon as function of deadline.

[7] Note that the standard deviation is orders of magnitude lower than the value of 1.0 reported for FILO.

5.6 Adaptation to Change

Figure 11 demonstrates DC-DRF adapting to changes in demand and changes in deadline. As before, we use the 8-second deadline suggested by section 5.5. In addition to earlier experiments, a random 5% of tenants change their demand by 5% at each control interval, to represent variations in demand that occur in the real world.

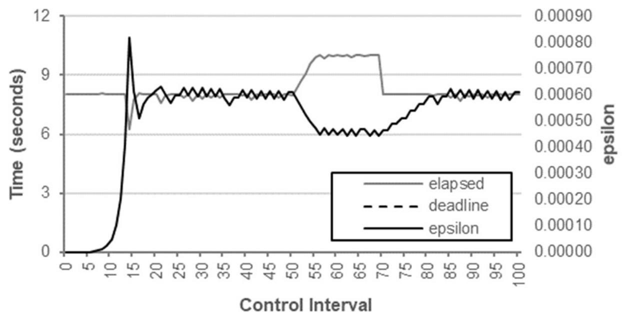


Figure 11 Adapting to variable demand or deadline.

Initially epsilon starts at zero, so the first control interval is terminated at the deadline triggering the search for epsilon in the second interval. Over the next 15 intervals, the DC-DRF outer loop expands its search window as the algorithm completes within the deadline. In the 15th control interval the elapsed time exceeds the deadline so DC-DRF starts to contract its search window. Once stable, DC-DRF adjusts epsilon by small amounts while keeping close to the deadline, in order to detect when to restart its search in reaction to variations in demand. At control interval 35 there is an abrupt change in demand when half the tenants change their demands by 50%, as may happen in response to a significant infrastructure failure; DC-DRF absorbs this with minor adjustment to the value of epsilon. At control interval 50 the deadline is increased to 10 seconds, for example due to a manual change by the datacenter provider, and DC-DRF adapts to this by commencing a new search for epsilon that completes at around interval 55. The deadline is changed back to 8 seconds at control interval 70, and again DC-DRF adapts by adjusting epsilon.

6. RELATED WORK

The foundational work of Dominant Resource Fairness (DRF) defined fairness properties upon which our work, and that of others, is based [17]. Extended DRF (EDRF) contributed a formalized specification of a DRF algorithm using a closed-form expression to calculate resource allocations in each round of the algorithm [32].

H-DRF investigated fairness for jobs in a shared compute cluster from the perspective of competing groups within the organizational structure (division, department, office, etc.) from which the jobs originate [10]. It concludes by recognizing that computing DRF for large inputs may be computationally expensive, and suggests this as an area for further research.

HUG used EDRF to investigate trade-offs between utilization and isolation guarantees at Public Cloud scale in specialized cases where every tenant must have non-zero demand for each and every resource [13]. For such scenarios they introduce “elastic demands” and add an additional phase following EDRF to provide work conservation for elastic demands. When elastic demands are present, the HUG approach for achieving work conservation is orthogonal to our approach, and could be applied after DC-DRF for those elastic demands that have been over-estimated.

Prior work utilized DRF at relatively small scales. Pulsar employed DRF to enable dynamic multi-resource differentiated service levels with work conversation for shared resources in a data center setting, running DRF in a central SDN-like controller [5]. Its scale was limited to Private Cloud due to the cost of calculating multi-resource allocations at its central controller. Filo used DRF to implement throughput guarantees in a distributed multi-tenant Cloud-based consensus service [31]. To decrease the time spent in calculating allocations, it used a distributed adaptation of DRF. Relatively to DC-DRF, its scale was small and the error incurred by distribution was relatively high.

Non-DRF approaches to resource allocation at cluster scale include a mix of priorities and quotas, as in Borg [43], and min-cost max-flow, as in Firmament [19].

Prior work has explored how to approximate max-min fairness. Awebuch et al. [6] designed an approximation algorithm up to a multiplicative ϵ factor, which works quite differently from our approach, based on a discretization of the set of permissible allocations on a logarithmic scale. However, being an approximation of max-min, their approach considers a single-resource only and is therefore not directly comparable to multi-resource DRF.

7. CONCLUSION

We have presented the DC-DRF algorithm, an adaptive approximation of EDRF designed to support centralized multi-resource allocation in bounded time. We have shown that a high-performance implementation of DC-DRF calculates multi-resource allocations at Public Cloud scale in practical time and with lower error than previous approaches. This removes a fundamental barrier to the deployment of multi-resource allocation in future cloud datacenters. A possible opportunity for further work would be to explore dynamic ways of detecting the set of resources used by a tenant, when this is not explicitly specified by either provider or tenant.

ACKNOWLEDGEMENTS

We would like to thank Jim Pinkerton, Andrew Blake, and Miguel Castro for encouraging this work, and Paolo Costa, Thomas Karagiannis, and Hitesh Ballani for reviewing early drafts of this paper. We are grateful to the anonymous reviewers for their diligence and the valuable feedback they have provided.

REFERENCES

[1] Apache Hadoop Yarn DRF scheduler.

[2] Windows Server 2016 technical preview storage quality of service.

[3] An Introduction to the Intel QuickPath Interconnect.

[4] Intel Intrinsics Guide.

[5] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *11th USENIX Symposium on Operating Systems and Design (OSDI)*, 2014.

[6] B. Awerbuch and Y. Shavitt. Converging to approximated max-min flow fairness in logarithmic time. In *17th Conference on Information Communications (INFOCOM)*, 1998.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM 2011 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2011.

[8] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[10] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *ACM Symposium on Cloud Computing 2013 (SoCC)*, 2013.

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, and J. e. a. Wu. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[12] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *49th International Symposium on Microarchitecture (MICRO)*, 2016.

[13] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems and Design (NSDI)*, 2016.

[14] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource Central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *26th ACM Symposium on Operating Systems (SOSP)*, 2017.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, A.-D. Kaynak, C.and Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[16] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *ACM SIGCOMM 2012 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[17] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, SoCC '18, October 11–13, 2018, Carlsbad, CA, USA Ian A. Kash, Greg O'Shea, and Stavros Volos 2011.

[18] D. Ghoshal, R. S. Canon, and L. Ramakrishnan. I/o performance of virtualized cloud environmentts. In *2nd International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2011.

[19] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[20] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[21] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *6th International Conference on Emerging Networking Experiments and Technologies (Co-NEXT)*, Co-NEXT '10, 2010.

[22] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *ACM SIGCOMM 2016 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[23] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of windows azure. In *19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos:

A platform for fine-grained resource sharing in the data center. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[25] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[26] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *9th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2009.

[27] I. A. Kash, G. O'Shea, and S. Volos. DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale. In *ACM Symposium on Cloud Computing 2018 (SoCC)*, 2018.

[28] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM 2014 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[29] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[30] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[31] P. J. Marandi, C. Gkantsidis, F. Junqueira, and D. Narayanan. Filo: Consolidated consensus as a cloud service. In *2016 USENIX Annual Technical Conference (ATC)*, 2016.

[32] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *ACM Conference on Electronic Commerce (EC)*, 2012.

[33] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *ACM SIGCOMM 2013 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[34] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *ACM SIGCOMM 2013 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[35] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *10th ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[36] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM 2013 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[37] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM 2015 Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[38] J. Schad, J. Dittrich, and J.-A. Quiane-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *33rd International Conference on Very Large Data Bases (VLDB)*, 2010.

[39] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[40] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[41] I. Stefanovici, E. Thereska, B. Schroeder, H. Ballani, A. Rowstron, and T. Talpey. Software-Defined Caching: Managing caches in multi-tenant data centers. In *ACM Symposium on Cloud Computing 2015 (SoCC)*, 2015.

[42] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *25th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[43] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems 2015 (EuroSys)*, 2015.

[44] E. Walker. Benchmarking amazon ec2 for high-performance scientific computing. ;LOGIN, 29(3):18–23, 2008.

[45] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *29th Conference on Information Communications (INFOCOM)*, 2010.

[46] H. Wang and P. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[47] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. *SIGCOMM Computer Communication Review*, 42(4):199–210, 2012