Hierarchical Scheduling for Diverse Datacenter Workloads

Arka A. Bhattacharya[1], David Culler[1], Eric Friedman[2], Ali Ghodsi[1], Scott Shenker[1], and Ion Stoica[1]

[1]University of California, Berkeley
[2]International Computer Science Institute, Berkeley

Abstract

There has been a recent industrial effort to develop *multi-resource hierarchical* schedulers. However, the existing implementations have some shortcomings in that they might leave resources unallocated or starve certain jobs. This is because the multi-resource setting introduces new challenges for hierarchical scheduling policies. We provide an algorithm, which we implement in Hadoop, that generalizes the most commonly used multi-resource scheduler, DRF [1], to support hierarchies. Our evaluation shows that our proposed algorithm, H-DRF, avoids the starvation and resource inefficiencies of the existing open-source schedulers and outperforms slot scheduling.

Categories and Subject Descriptors

D.4.1 **[Process Management]:** Scheduling;

Keywords

Multi-resource, Data Center, Fairness

1 Introduction

Cloud computing frameworks tailored for managing and analyzing big datasets are powering ever larger clusters of computers. Efficient use of cluster resources is an important cost factor for many organizations, and the efficiency of these clusters is largely determined by scheduling decisions taken by these frameworks. For this reason, much research has gone into improving datacenter schedulers [2, 3, 4, 5, 6, 7, 8]. A key feature of all production cloud schedulers is *hierarchical scheduling*, which enables scheduling resources to reflect organizational priorities. Most production schedulers today support hierarchical scheduling (*e.g.,* Hadoop Capacity Scheduler [6] and Hadoop Fair Scheduler [9]). As an example of hierarchical scheduling, an organization (see Figure 1) might dedicate 60% of its resources to the ad department, and 40% to the dev department. Within each department, the resources are further split, for example 70% for production jobs, and 30% for test jobs. The key feature of hierarchical scheduling—which is absent in flat or non-hierarchical scheduling—is that if some node in the hierarchy is not using its resources they are redistributed among that node's sibling nodes, as opposed to all leaf nodes. For example, if there are no test jobs in the ads department, those resources are allocated to prod jobs in that department. Most organizations that we have spoken to—including Facebook, Yahoo, and Cloudera—use hierarchical scheduling to allocate resources according to organizational structures and priority concerns.

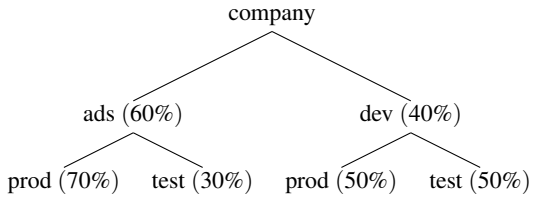


Figure 1: Simple Organizational Hierarchy.

Recently, there has been a surge of research on multi-resource scheduling [10, 11, 12, 13]. It has been shown that workloads in data centers tend to be diverse [14], containing a mix of jobs that are CPU-intensive, memory-intensive, or I/O intensive [15, 16]. Therefore, efficient scheduling in datacenters requires taking multiple resource types into account. Otherwise,

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1. http://dx.doi.org/**10.1145/2523616.2523637**

the scheduler might make allocations based on irrelevant resources (*e.g.,* CPU), ignoring the actual resource needs of jobs (*e.g.,* memory), ultimately leading to poor performance isolation and low throughput for jobs. To handle multi-resource workloads, a new scheduler, called *Dominant Resource Fairness (DRF)*, was recently proposed [1] and shipped with the open source resource manager Mesos [17]. Since DRF's introduction, a string of follow-up papers have analyzed, improved, and extended DRF [10, 11, 12, 13]. Unfortunately, DRF does *not* have support for hierarchical scheduling.

The need for multi-resource scheduling with the additional requirement of supporting hierarchical scheduling is crucial and has, therefore, led to an industrial effort to provide multi-resource hierarchical scheduling. The Capacity scheduler [18] was rewritten for Hadoop Next-Generation (YARN) [19] to provide multi-resource DRF allocations. Furthermore, work is underway to extend the Fair Scheduler to support DRF [20]. These efforts have been initiated and led by the companies Horton-Works and Cloudera independently of this work.

However, the combination of hierarchical and multi-resource scheduling brings new challenges that do not exist in the traditional single-resource hierarchical setting, and are not fully addressed by these production efforts. Naive implementations of multi-resource hierarchical schedulers can lead to the starvation of some jobs, certain sub-trees in the hierarchy not receiving their fair resource share, or some resources being left unallocated (§3). Identifying the shortcomings is a start, but coming up with an algorithm that avoids these pitfalls is a larger challenge.

In this paper we introduce an online multi-resource scheduler, called *H-DRF*, that supports hierarchical scheduling. H-DRF guarantees that each node in the hierarchy at least gets its prescribed fair share of resources, regardless of how others behave. We refer to this as the *hierarchical share guarantee*, which is an important isolation property. It implies that no job will starve, but also that each group in the organization gets its allotted share, two desirable outcomes not provided by other approaches. Finally, H-DRF is group-strategyproof, which means that no group of users can increase their useful allocation as a whole by artificially inflating or changing their resource consumption. This property is trivially satisfied when there is a single resource, but becomes non-trivial to satisfy in multi-resource settings.

We have implemented our H-DRF algorithm for Hadoop 2.0.2-alpha (YARN) [21]. Evaluations show that H-DRF outperforms the existing implementation of the Capacity Scheduler [6] in terms of efficiency and job starvation. Also, through simulations on a Facebook cluster trace and example workloads on our prototype, we show that H-DRF outperforms hierarchical slot schedulers by better packing of jobs and achieving a higher throughput.

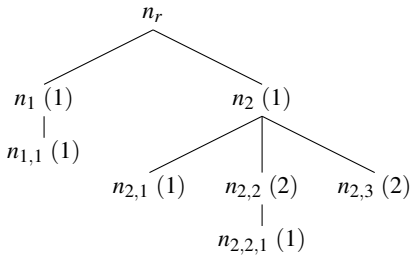


Figure 2: Example hierarchy, with node notation and weights in parenthesis.

2 Background

We begin by providing background on hierarchical scheduling and multi-resource fairness.

2.1 Hierarchical Scheduling

Notation. A hierarchical scheduler is configured with a weighted tree, such that each node in the tree has a positive weight value (see Figure 2). The weights denote relative importance and do not have to sum to 1. The leaves in the tree denote the *jobs* (or users)[1] that are ultimately to be allocated resources, whereas the internal nodes represent organizational or hierarchical groups. We assume each job/user submits a set of *tasks*, whose demands are not necessarily known in advance.

A node in the tree is denoted n_L where L is a list of numbers that describe how the node can be found when starting at the top of the tree and going down, left to right. The root node of the tree is n_r. For example $n_{2,1,4}$ is found by starting at the root, picking its second (from left) child, picking that child's first child, and that child's fourth child. We denote the weight of node n_L's weight by w_L. The parent of a node is given by $P()$, *i.e.,* $P(\langle 2,2,1\rangle) = \langle 2,2\rangle$. We write $P^i(L)$ to refer to the i:th predecessor of node n_L, *i.e.,* $P^3(L) = P(P(P(L)))$. Similarly, the set of children of a node is given by $C()$. We mostly only care about the set of nodes that are currently *demanding* more resources. A leaf node is demanding if it asks for more resources than are allocated to it, whereas an internal node is demanding if any of its children are demanding. The function $A()$ takes a set of nodes in a tree and returns the subset of those nodes that currently are demanding.

Hierarchical Share Guarantee. The main idea behind hierarchical scheduling is to assign to each node in the tree some guaranteed share of the resources. A node in the hierarchy is entitled a fraction of resources from

[1] We use the terms *job* and *user* interchangeably to denote a leaf node in the hierarchy

its parent proportional to the ratio of its weight to that of its demanding siblings' including itself. That is, a node n_L is guaranteed to get a fraction of resources from its parent $n_{P(L)}$ of at least:

$$\frac{w_L}{\sum_{i \in A(C(P(L)))} w_i}$$

In the example in Figure 2, assume there are 480 slots[2] in a cluster, and that all nodes are demanding. Then, n_1 and n_2 should get 240 slots each, as they have the same weight, 1. One level down, the children of n_1 should get a number of slots from their parent's allocation proportional to their weights, i.e., $n_{2,1}$ should get 48 ($= 240/5$), while $n_{2,2}$ and $n_{2,3}$ should get 96 ($= 240 \times 2/5$) each.

The above hierarchical share guarantee captures a key feature that hierarchical scheduling provides, called *sibling sharing*, which enables resources to stay within a sub-organization in the hierarchy when jobs finish within that sub-organization. Sibling sharing guarantees that if a node in the tree leaves, its resources are given to its demanding siblings in proportion to their weights. If it has no demanding siblings, the resources are given to its parent, which recursively gives it to the parent's siblings. For example, if all nodes are demanding and $n_{2,3}$ leaves, then its 96 slots are split 32 and 64 to $n_{2,1}$ and $n_{2,2}$, respectively. Unlike flat scheduling, nothing is given to n_1 or its children, unless there are no demanding nodes in the subtree of n_2.

2.2 Dominant Resource Fairness (DRF)

Many datacenters exhibit a diverse workload, containing a mix of jobs that can be CPU-intensive, memory-intensive, or I/O intensive [15, 1]. A multi-resource fairness mechanism known as Dominant Resource Fairness (DRF) [1] introduced the concept of a job's (user's) *dominant resource*, which is the resource that the job needs the highest share of. DRF seeks to equalize the dominant shares across all jobs (users) in a cluster. A job's *dominant share* is simply the share of its dominant resource that it is currently allocated.

For example, if a cluster has 100 GB of memory, and 100 CPUs and each of a job's task requires 3 GB of memory and 2 CPUs to run, memory is the job's dominant resource because each of its tasks requires 3% of the entire cluster memory whereas it only requires 2% of the entire cluster CPUs.

If the same cluster runs two jobs whose task requirements are ⟨3GB, 2CPUs⟩, and ⟨2GB, 3CPUs⟩, each job receives a dominant share of $\frac{60}{100} = 0.6$ (or 60%) as shown in Figure 3. This is unlike single-resource fairness, in which the sum of all jobs' dominant shares can never be more than 1.0 (or 100%). We call this phenomenon of jobs having complimentary dominant resources leading to the sum of dominant shares being more than 1.0 as *dovetailing*.

Note that one of the important aspects of multi-resource fairness, as identified in DRF [1], is that some job might have zero demand for some resources. For instance, some jobs might not need to use GPUs or specialized hardware accelerators. This is likely to increase as more resources are accounted for by schedulers.

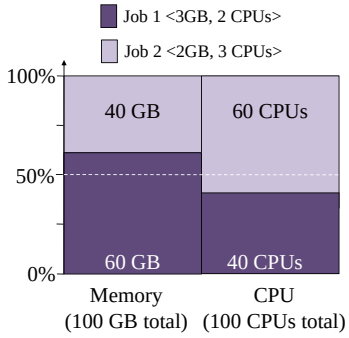


Figure 3: DRF allocation shares for two jobs whose tasks have the resource requirements shown in angular brackets. Note that the share of Job 2's dominant resource (CPU) is equal to the share of Job 1's dominant resource (Memory)

.

3 Hierarchical DRF (H-DRF)

We first adapt the definition of static dominant share allocation in DRF [1] to support hierarchies (§3.1). We shall call such allocations *Static H-DRF* allocations. However, achieving the static H-DRF allocations in a dynamic system is non-trivial. §3.2 and §3.3 describe the shortcomings of two existing approaches that could be used to achieve the static H-DRF allocations in a dynamic setting – *Collapsed hierarchies* and *Naive H-DRF*. Finally, we propose our solution – *Dynamic H-DRF* in §3.4.

Although we assume in this section, for simplicity, that all nodes have the same weight, our discussion can be generalized to multiple weights in a straightforward fashion (*c.f.,* [1]).

3.1 Static H-DRF Allocations

We develop a static version of DRF to handle hierarchies. The static definition is not meant to be used for scheduling, but rather to define the allocation. The dynamic algorithm is instead used to schedule resources to achieve the static allocation.

The main aim of static H-DRF allocations is to equalize the dominant resource share between each pair of sibling nodes in the hierarchy. Given a set of nodes(jobs)

[2] A slot is a fixed fraction of a server, *e.g.,* 1 core, 4 GB memory, and 10 GB disk [9].

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
n_r ▷ root node in hierarchy tree
$C(n)$ ▷ children of any node n
$P = \langle P^1(n_i), P^2(n_i) \cdots \rangle$ ▷ List of n_i's parents
$s_i \ \ (i = 1 \ldots n)$ ▷ node n_i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ \ (i = 1 \ldots n)$ ▷ resources given to node n_i, initially 0

$P = \emptyset$
$n_i = n_r$
while resources exist to allocate more tasks **do**
  while n_i is not a leaf node (job) **do**
    $P = P \cup \langle n_i \rangle$
    n_j = node with lowest dominant share s_j in $C(n_i)$, which also has a task in its subtree that can be scheduled using the current free resources in the cluster
    $n_i = n_j$
  $D_i = \frac{\varepsilon}{\max_j\{T_{i,j}\}} T_i$, s.t. T_i is n_i's task demand vector
  $C = C + D_i$ ▷ update consumed vector
  for each node k in $n_i \cup P$ **do**
    $U_k = U_k + D_i$ ▷ update allocation vectors
    $s_k = \max_{j=1}^{m}\{U_{i,j}/r_j\}$ ▷ update Dominant Resource shares

Algorithm 1: Static H-DRF Allocation

with demands and a set of resources, static Hierarchical DRF (H-DRF) starts with every job being allocated zero resources, and then repeatedly increasing each job's allocation with a thin sliver (ε) of resources until no more resources can be assigned to any node. The final assignment constitutes the static H-DRF allocation for the given nodes' demands and total available resources.

More precisely, at each moment, record the amount of resources assigned to each leaf node (job) in the hierarchy. Internal, non-leaf, nodes (sub-organizations) are simply assigned the sum of all the resources assigned to their immediate children. Start at the root of the tree, and traverse down to a leaf, at each step picking the *demanding* (*c.f.,* §2.1) child that has the smallest dominant share. In case of tie, randomly pick a node. Then allocate the leaf node an ε amount of its resource demands, *i.e.,* the resource demand vector of that node is resized such that that node's dominant share is increased by ε.[3] Algorithm 1 shows pseudocode for how static allocations are computed.

For example, the H-DRF allocation for the hierarchy in Figure 4(a) can be computed as follows in a system with 10 CPUs and 10 GPUs. Node $n_{1,1}$ is first assigned $\langle \varepsilon, 0 \rangle$. This makes n_1's dominant share ε. Thereafter, n_2 is traversed, since it has a lower dominant share than n_1, picking $n_{2,1}$, which is assigned $\langle \varepsilon, 0 \rangle$. Next, $n_{2,2}$ assigned $\langle 0, \varepsilon \rangle$. This puts n_2 at $\langle \varepsilon, \varepsilon \rangle$, which gives it a dominant share of ε. This process is repeated by assigning ε tasks until some resource is completely exhausted, which in this case will be CPU. At this point, nodes $n_{1,1}$ and $n_{2,1}$ become non-demanding as they cannot be allocated more resources, as all 10 CPU resources have been allocated. Thereafter, the process continues by assigning $\langle 0, \varepsilon \rangle$ tasks to $n_{2,2}$ until all GPUs have been assigned. This defines the H-DRF allocation to be $\langle$5 CPUs,0 GPUs$\rangle$ to $n_{1,1}$ and $n_{2,1}$ each, and $\langle$0 CPUs,10 GPUs$\rangle$ to $n_{2,2}$ (Figure 4(b) depicts the allocation).

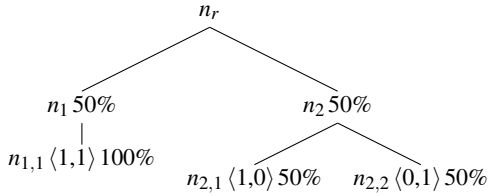
However, in a cluster, tasks finish and new ones are launched. Re-calculating the static H-DRF allocations for each of the leaves from scratch at the arrival of each new task is computationally infeasible. The following subsections will formulate an algorithm that achieves the static-HDRF allocations in such dynamic settings.

3.2 First attempt: Collapsed Hierarchies

One well-known approach, which we call *collapsed hierarchies* [22], converts a hierarchical scheduler into a flat one. The idea is to take the hierarchical specification and compute what the corresponding weights for each leaf node would be if the hierarchy was flattened. These weights are then used with a flat scheduler, such as the original weighted DRF algorithm [1]. Each time jobs are added, removed, or change their demand-status, the weights are recalculated. For simplicity, we ignore how recalculation is done as this approach breaks down even without recalculation.

This approach *always* works when only one resource is involved. Interestingly, the approach fails to work when multiple resources are scheduled. In particular, it will violate the hierarchical share guarantee for internal nodes in the hierarchy if they *dovetail*.

Consider a slight modification to the example hierarchy in Figure 4(a), where $n_{1,1}$ instead wants to run tasks with demands $\langle$1 CPU, 1 GPU$\rangle$:



The hierarchy is flattened by assigning to each node a weight that corresponds to the product of its weighted shares in the hierarchy from the leaf to the root. Nodes $n_{2,1}$ and $n_{2,2}$ are each assigned $0.5 \times 0.5 = 0.25$, since they each are entitled to half of their parents allocation,

[3] e.g a if a node demands $\langle 1CPU, 2GPU \rangle$, with equal amounts of both resources in the cluster, the node is allocated $\langle \frac{\varepsilon}{2} CPU, \varepsilon GPU \rangle$

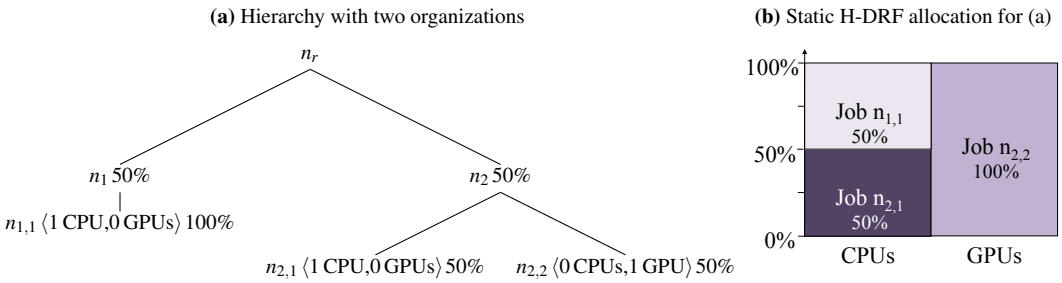
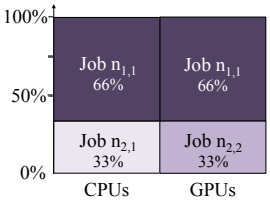


Figure 4: Simple hierarchy and its static H-DRF allocation

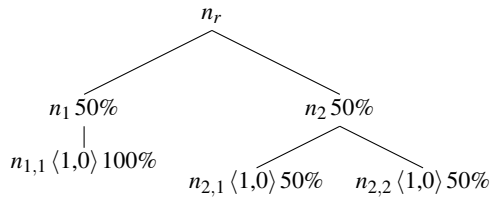
which is entitled to half of the cluster. Node $n_{1,1}$ is similarly assigned $0.5 \times 1.0 = 0.5$. These weights accurately capture the share guarantee for each leaf node.

We now run the original non-hierarchical DRF algorithm [1] configured with these weights and get the following allocation:



Since $n_{1,1}$ has twice the weight of the rest of the leaves, DRF will increase its dominant share at twice the rate of those other nodes. Both resources, CPU and GPU, will then saturate when $n_{1,1}$ is allocated $\frac{2}{3}$ of the CPUs and GPUs, and $n_{2,1}$ and $n_{2,2}$ are allocated $\frac{1}{3}$ of their respectively demanded resource. While each leaf node's hierarchical share guarantee has been satisfied, the internal node n_2 has only gotten 33% of resources as opposed to its entitled 50%, violating the hierarchical share guarantee.

The above problem is new in the multi-resource setting. Consider a modification to the problem that turns it into a single-resource problem. Assume that $n_{2,2}$ would have demanded $\langle 1$ CPU,0 GPUs$\rangle$, thus making all jobs only demand CPU:



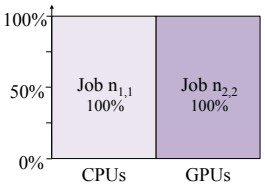
Then the above method would allocate 50% of the CPUs to $n_{1,1}$, and 25% each to jobs $n_{2,1}$ and $n_{2,2}$, satisfying the hierarchical share guarantees for all nodes in the hierarchy. The problem in the multi-resource setting is that dove-tailing of resource demands, *i.e.,* that jobs have complementary resource demands ($n_{2,1}$'s $\langle 1,0 \rangle$ and $n_{2,2}$'s $\langle 0,1 \rangle$ in the example) "punishes" the parent nodes.

3.3 Second Attempt: Naive H-DRF

We now turn to a natural adaptation of the original DRF to the hierarchical setting and show that it can lead to starvation. In particular, we show how the hierarchical share guarantee is violated for leaf nodes. In fact, the current Hadoop implementations, which implement hierarchical DRF [18], take this approach and, hence, suffer from starvation as we show in the evaluation (§5.2). Consider a dynamic algorithm—which we call *Naive H-DRF*—that simply assigns a task in a similar manner to how the static H-DRF allocation is computed each time resources become available *i.e.,* traverse the tree from root to leaf, at each step pick the demanding child with smallest dominant share, until a leaf node (job) is reached and allocate one task to that leaf node.

To see how starvation can occur, consider the example hierarchy given in Figure 4(a), and assume 10 CPUs, 10 GPUs, and three demanding leaf nodes.

The algorithm will initially allocate 5 CPUs each to $n_{1,1}$ and $n_{2,1}$, and 10 GPUs to $n_{2,2}$, as illustrated in Figure 4(b). The problem occurs when when tasks finish and new ones are launched. Consider when job $n_{2,1}$ finishes a task, which has dimension $\langle 1$ CPU,0 GPUs$\rangle$. The dynamic algorithms traverses the tree, but notes that the internal node n_2 has a dominant share of 100% (10 GPUs out of 10 total). It will therefore pick n_1 and finally allocate a task to $n_{1,1}$. This will repeat itself until job $n_{2,1}$ is completely starved and 10 CPUs have been allocated to $n_{1,1}$ and 10 GPUs to $n_{2,2}$. At this point, the algorithm has equalized and allocated a dominant share of 100% to each group n_1 and n_2, but has violated the hierarchical sharing guarantee for node $n_{2,2}$, which is allocated zero resources. This leads to the following allocation:



3.4 Our solution : Dynamic Hierarchical DRF

We now derive the Dynamic Hierarchical DRF algorithm, H-DRF. We do so in two steps, combining two ideas that together achieve static H-DRF allocations, do not suffer from starvation, and satisfy the hierarchical share guarantee.

Rescaling to Minimum Nodes. Starvation happens in the Naive H-DRF algorithm because of the way the algorithm attributes resource consumption to internal nodes. In the example of Figure 4(b), node n_2 is attributed to have a dominant share of 100% since one of its jobs ($n_{2,2}$) has a dominant share of 100%. Naive H-DRF keeps punishing $n_{2,1}$ as long as $n_{2,2}$ has a higher dominant share than $n_{1,1}$. We therefore change how resource consumption is attributed at internal nodes.

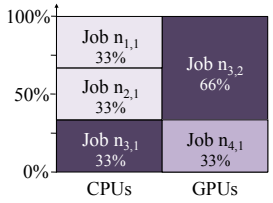
To compute the resource consumption of an internal node, proceed in three steps. First, find the *demanding* child with minimum dominant share, M. Second, rescale every child's resource consumption vector so that its dominant share becomes M, *i.e.,* each element of a resource consumption vector with dominant share D is multiplied with $\frac{M}{D}$. Third, add all the children's rescaled vectors to get the internal node's resource consumption vector.

Consider, for example, how n_2's resource consumption vector is computed in the example given in Figure 4. First, the children of n_2 have dominant shares 0.5 and 1.0, respectively, yielding the minimum $M = 0.5$. Second, we rescale each child's whole resource consumption vector to have a dominant share of 0.5. This means that $n_{2,1}$'s vector is rescaled to $\frac{0.5}{0.5} \times \langle 0.5, 0 \rangle = \langle 0.5, 0 \rangle$. Similarly, $n_{2,2}$ is rescaled to $\frac{0.5}{1.0} \times \langle 0, 1 \rangle = \langle 0, 0.5 \rangle$. Third, the internal node n_2's resource consumption is the sum of those vectors, *i.e.,* $\langle 0.5, 0.5 \rangle$, yielding a dominant share of 0.5 for n_2.

The above method avoids the previously given starvation scenario. Consider the allocation given by Figure 4(b). If $n_{2,1}$ finishes a task, it will be allocated a new task since its dominant share will go below 50%, making the parent's dominant share go—through rescaling—below 50% as well. Similarly, if any other job finishes a task, the resources are offered back to it.

While rescaling helps the above example, it alone is not enough to achieve static H-DRF allocations, as minimization can give an unfair advantage to certain nodes as the following example shows. The hierarchy in Figure 5(a) has the static H-DRF allocation given by Figure 5(b). The first resource is the most demanded and will saturate first. At that point, every leaf is allocated $\frac{1}{3}$ of its dominant share. Thereafter, only leaves $n_{3,2}$ and $n_{4,1}$ can still run more tasks, so the rest of the second resource is split evenly between them.

H-DRF with rescaling only (the algorithm described thus far) will, however, yield the following allocation:

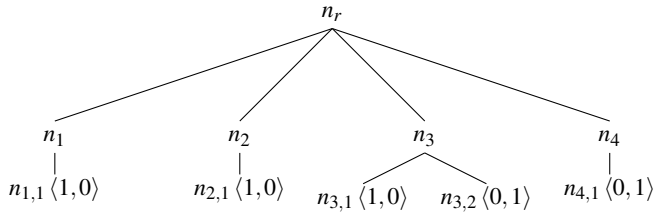


It allocates tasks similarly until the first resource becomes saturated. But rescaling will always normalize $n_{3,2}$'s dominant share to that of $n_{3,1}$, *i.e.,* to $\frac{1}{3}$. As soon as another task is assigned to $n_{4,1}$, n_4's dominant share will be higher than n_3's, resulting in all remaining GPU resources being repeatedly allocated to $n_{3,2}$. Thus, in the final allocation $n_{3,2}$ gets $\frac{2}{3}$ of GPUs, while $n_{4,1}$ gets only $\frac{1}{3}$ of the GPUs. We address this problem next.

Ignoring Blocked Nodes. When rescaling to attribute internal node consumption, dynamic H-DRF should only consider non-blocked nodes for rescaling. A leaf node (job) is *blocked* if either (i) any of the resources it requires are saturated, or (ii) the node is non-demanding, *i.e.,* does not have more tasks to launch. Recall, that a resource is saturated when it is fully utilized. An internal node is blocked if all of its children are blocked. The three aforementioned steps required to compute an internal node's resource consumption vector are modified as follows. First, pick the minimum dominant share, M, among non-blocked nodes. Second, only every non-blocked node's resource consumption vector is rescaled such that its dominant share is M. Third, all nodes'—blocked as well as non-blocked—vectors are added to get the parent's resource consumption vector. Furthermore, we ignore saturated resources when computing the dominant share of any internal node.

The above modification will now ensure that the example in Figure 5(a) is correct, *i.e.,* the algorithm will yield the static H-DRF allocation given in Figure 5(b). To see this, the algorithm will behave the same until the first saturation point, as there will be no blocked jobs. When the first resource saturates, every job has a dominant share of $\frac{1}{3}$. Since $n_{3,1}$ is blocked on CPU, its dominant share will not be used during rescaling. Thus, n_3's dominant share will thereafter be equal to $n_{3,2}$'s. Therefore, the remainder of GPUs will be equally allocated to $n_{3,2}$ and n_4, yielding the static H-DRF allocation in

(a) Hierarchy for which rescaling breaks.



(b) Static H-DRF allocation for (a)

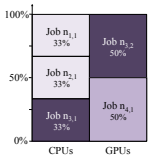


Figure 5: Hierarchy that breaks rescaling and its static H-DRF allocation. The demand vector $\langle i, j\rangle$ represents i CPUs and j GPUs.

Figure 5(b).

Just ignoring blocked nodes, without rescaling, will not be sufficient to achieve static H-DRF allocations. To see this, consider the example in Figure 4(a), where job $n_{2,1}$ eventually gets starved because its resources are given to $n_{1,1}$ in a dynamic system. Recall that the problem was that the internal node n_2 was attributed to have a dominant share of 100%. Ignoring blocked nodes indeed will ensure that $n_{2,1}$ is not starved, as $n_{2,2}$ is blocked, giving n_2 a dominant share equal to that of $n_{2,1}$. If we, however, modify the example so that $n_{2,1}$ demands $\langle 1, \varepsilon\rangle$, ignoring blocked nodes alone will no longer suffice. Each time $n_{2,1}$ finishes a task $\langle 1, \varepsilon\rangle$, some amount of both resources is released. Thus, $n_{2,2}$ is no longer blocked as there are no saturated resources. Thus, n_2's dominant share will again be close to 100%. Rescaling will, however, remedy the problem as $n_{2,2}$ is scaled back to $n_{2,1}$, ensuring that CPUs are split equally between $n_{1,1}$ and $n_{2,1}$.

Final Dynamic H-DRF Algorithm Algorithm 2 puts together the ideas that were derived in this section, *i.e.,* naive H-DRF modified with rescaling to minimum nodes, and ignoring blocked nodes. It begins by recursively computing the rescaled dominant shares based on the demanding leaves (Algorithm 3) and then applies the same allocation procedure as in the static version based on the rescaled dominant shares (Algorithm 4). Note that the recompilation of the rescaled dominant shares can be simplified when the set of available resources has not changed; however for simplicity of presentation we ignore this below.

To compare the static and the dynamic H-DRF, consider a simple dynamic model of system behavior. Begin with no resources assigned and run the dynamic H-DRF where tasks can complete at arbitrary times and whenever there are resources available they are iteratively allocated according to dynamic H-DRF. We assume that tasks are small since allocations are done by small slices. We then compare dynamic H-DRF with the static allocation at times when there are no excess resources, except those which are not useful to any of the leaves. Under this model, we can show the following.

$R = \langle r_1, \cdots, r_m\rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m\rangle$ ▷ current consumed resources
W resources to allocate ▷ Assumption:$R - C > W$
Y set of nonzero resources in W
A (demanding), set of leaf nodes that use only resources in Y or parents of demanding nodes
n_r ▷ root node in hierarchy tree
$C(n)$ ▷ children of any node n
s_i $(i = 1 \ldots n)$ ▷ dominant shares
$U_i = \langle u_{i,1}, \cdots, u_{i,m}\rangle$ $(i = 1 \ldots n)$ ▷ resource consumption of node i
Recompute s**:** $UpdateS(n_r)$
Allocate the resources: Alloc(W)

Algorithm 2: Dynamic H-DRF Algorithm

function (recursive) $UpdateS(n_i)$
if n_i is a leaf node **then**
  $s_i = \max U_{ij}/R_j$ for $j \in Y$
  return U_i
else
  Q = set of U_j's from $UpdateS(n_j)$ for children of n_i
  f = minimum dominant share from Q restricting to nodes in A and resources in Y
  Rescale demanding vectors in Q by f
  U_i = sum of vectors in Q
  $s_i = \max U_{i,j}/Rj$ for $j \in Y$
return U_i

Algorithm 3: Dynamic H-DRF Rescaling Function

Theorem 1 *Static H-DRF allocations agree with dynamic H-DRF allocations whenever resources are fully allocated.*

To see why this result holds, we first note that by the monotonicity of the static allocations of resources (discussed in more detail after Theorem 2), for the initial allocation of resources, before any tasks complete, the

function *Alloc*(W)
$n_i = n_r$
while n_i is not a leaf node (job) **do**
    n_j = node with lowest dominant share s_j in $C(n_i)$, which also has a task in its subtree that can be scheduled using W
    $n_i = n_j$
$D_i = \frac{W_i}{\max_j\{T_{i,j}\}} T_i$, s.t. T_i is n_i's task demand vector
$C = C + D_i$ ▷ update consumed vector
$U_i = U_i + D_i$ ▷ update leaf only

Algorithm 4: Dynamic H-DRF Allocation Function

dynamic and the static version algorithms are identical, as no rescaling is necessary.

To complete the analysis we need to show that when a (very small) task completes it is reallocated to the leaf that completed it. It is clear that this occurs for any task that has the same set of resources as the last task allocated in the static H-DRF. It is also true for other completed tasks.

To see why, consider a leaf that was one of the first to be blocked in the static H-DRF allocation by the complete allocation of a resource r. Define s' to be the dominant resource shares for the nodes at that instant, s to be the shares after the release of the task and s^* the shares at the completion of the static H-DRF. Since the static H-DRF allocation is monotonic, $s' \leq s*$. However, if we consider the final allocation under static H-DRF but assume that r is demanding, then by the rescaling rule the dominant resource shares will be s'. This implies that s is smaller than s' for all parents of the leaf that completed the task and unchanged for all other nodes, which implies that the algorithm will allocate the task back to the node that released it. One can apply this argument inductively to show that it works in general.

4 Allocation Properties

The previous section showed that a simple approach to supporting hierarchies in DRF failed to provide certain guarantees and proposed H-DRF, which satisfies those. This section discusses several important properties of H-DRF and provides intuitions behind these properties.

The reasoning of H-DRF will be based on the static version of H-DRF which is easier to analyze and as we discussed in the previous section, the static and dynamic versions of H-DRF lead to the same allocations. The key idea behind the analysis of the static H-DRF allocation procedure is that it can be viewed as a water filling algorithm, with multiple types of "water" and carefully adjusted flow rates at each node. Then we use the monotonicity of water filling, since as the algorithm runs, the allocation of leaf nodes is increased monotonically. Since dominant shares are being equalized whenever possible, the sooner a leaf becomes blocked the lower its dominant share will be.

The static H-DRF algorithm does not depend on the scaling of a leaf's requirements, ⟨3GB, 2CPUs⟩ is treated the same as ⟨6GB, 4CPUs⟩, so we can simplify our analysis by assuming that the requirement for every dominant resource is 1 and also that the total amount of each resource is 1.

4.1 Hierarchical Share Guarantees

In the previous section we saw job starvation with naive H-DRF and that both naive H-DRF and Collapsed Hierarchies violated the group guarantees for internal nodes in the hierarchy. The Hierarchical Share Guarantee (defined in Section 2) precludes such violations. We now show that static (and hence dynamic) H-DRF satisfies these.

Theorem 2 *Static H-DRF allocations satisfy the Hierarchical Share Guarantee property.*

This guarantee implies that the allocation satisfies the so-called Sharing incentive, which implies that every node prefers the H-DRF allocation to splitting the entire system among the nodes. For example, given the hierarchy in Figure 4, both n_1 and n_2 prefer the H-DRF allocation to receiving half of the total resources.

To see why this result is true, consider a modified problem where we add a single additional resource with demand 1 to all the leaves. Now, consider running the static algorithm until the first time where this resource is fully utilized. At this instant it is easy to see recursively that every node has received exactly its hierarchical share guarantee. Now, compare this allocation to that without the extra resource. Until the point where the extra resource is fully utilized, the allocations on the other resources are unchanged. Thus, by monotonicity of the water filling, each node will end up with as good or better an allocation than the modified one.

4.2 Group Strategyproofness

Somewhat surprisingly, the original DRF [1] paper showed that in the multi-resource setting users can manipulate schedulers by artificially using more resources. This is a problem that does not occur in single-resource settings, but is an important concern in the multi-resource setting.

In the context of hierarchical scheduling, other manipulations become natural. For example, users within an organization could collude, coordinating their manipulations. To prevent these problems we require that the allocation mechanism satisfy group strategyproofness, a hierarchical extension of group strategyproofness.

Definition An allocation mechanism is group strategyproof if no group of users can misrepresent their resource requirements in such a way that all of them are weakly better off[4] and at least one of them is strictly better off.

Theorem 3 *H-DRF allocations satisfy group strategyproofness.*

To understand this result, again consider stopping the static H-DRF algorithm at the time where the first resource becomes saturated. If one of the leaves that is blocked at this point were to decrease its requirement for some of its non-dominant resources then the blocking time would not change and that leaf would receive less of the resources with the decreased requirements leading to a decrease in number of jobs. Alternatively, the leaf node could try to increase its requirement for some of its non-dominant resources. Two cases are possible in this scenario – either this would not change the first blocking time and the leaf would get more resources, but would not be able to utilize them as their allocation of their dominant resource would be unchanged, or alternatively, one of these non-dominant resources might block first, but then the leaf would get even less of their dominant resource. Thus we see that one of the first blocked leaves can not increase its allocation by lying about its requirements. One can also see that no group of first blocked leaves can lie so that all get better allocations by the same reasoning. Lastly, combining the recursive nature of the algorithm with the time monotonicity of the water filling it is straightforward to show that the same reasoning applies to all leaves independent of their first blocking time.

4.3 Other properties

The previous sections covered the most important properties of H-DRF. Here we briefly mention two other properties: recursive scheduling and population monotonicity.

Recursive Scheduling.

Definition Recursive scheduling: It should be possible to replace any sub-tree in the hierarchy tree with another scheduler.

Recursive scheduling is useful in many ways and allows one to modify a procedure by changing the algorithm on specified nodes of the tree. This allows any sub-organizations the autonomy to use alternative allocation rules *e.g.,* FIFO, Fair, *etc* if they so wish.

Theorem 4 *H-DRF allocations satisfy Recursive scheduling.*

[4]By weakly better off we mean that no one is worse off.

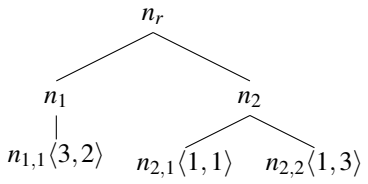


Figure 6: Example showing that H-DRF does not satisfy population monotonicity. The demand vector $\langle i, j\rangle$ represents i CPUs and j GPUs.

The replacement of H-DRF at some internal node with some other allocation rule may clearly impact the sharing incentive and strategy proofness for all children of that node, but does not in the other parts of the tree.

Population Monotonicity.

Definition Population monotonicity: Any node exiting the system should not decrease the resource allocation to any other node in the hierarchy tree.

In other words, in a shared cluster, any job completion should not affect the dominant resource share of any other node or job. This is important because job completions are frequent in a large cluster. Unfortunately, this does not hold for H-DRF. But this is because population monotonicity is incompatible with the share guarantee.

Theorem 5 *H-DRF allocations do not satisfy Population monotonicity.*

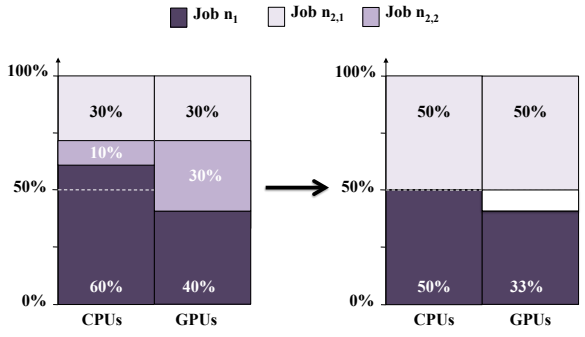


Figure 7: Resource allocations to the leaf nodes in Figure 6, when all three leaf nodes - $n_{1,1}$, $n_{2,1}$ and $n_{2,2}$ are demanding (left), and when only $n_{1,1}$ and $n_{2,1}$ is demanding (right)

Thus, when a user or organization in the hierarchy tree becomes non-demanding, other nodes may receive a lower share of their dominant resource. Let us consider the simple hierarchy shown in Figure 6. HDRF gives

n_1 60% share of its dominant resource (CPUs) when all three leaf nodes are demanding (Figure 7). If $n_{2,2}$ were to become non-demanding, the dominant resource share of n_1 reduces to 50%, since both queues n_1 and n_2 now have the same dominant resource. The intuition behind why H-DRF is not population monotonic is that the dominant resource of an internal node might change when any of it's children nodes becomes non-demanding resulting in reduction of dovetailing.

5 Evaluation

We evaluate H-DRF by deploying a prototype on a 50-server EC2 cluster running hadoop-2.0.2-alpha [21] and through trace-driven simulations. We modify hadoop-2.0.2-alpha to add support for GPUs as a resource. The H-DRF implementation in our prototype is single-threaded and centralized. H-DRF maintains a headroom equal to the size of the largest task on each server, to ensure that large tasks do not get starved out.

We first demonstrate fair sharing in H-DRF through a simple experiment. Then, we compare the performance of H-DRF to the Capacity Scheduler provided in Hadoop [6]. Finally, we compare the job performance metrics of H-DRF to that of hierarchical slot-based fair schedulers ([9]) through an example workload on our prototype, and through a simulation of a 10-day Facebook trace.

Notation: Jobs are only submitted to the leaf nodes in the shown hierarchies. Every job has the same resource requirements for all its tasks. In this section, the notation $\langle i, j, k \rangle$ denotes i GB of memory, j CPUs and k GPUs.

5.1 Hierarchical Sharing

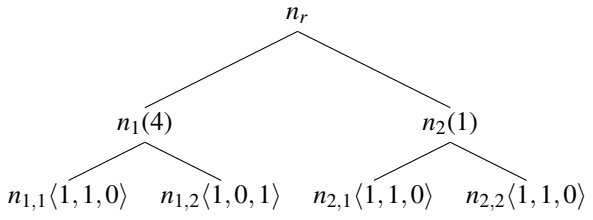


Figure 8: Hierarchy to demonstrate the working of H-DRF and fair allocation of resources

We illustrate fair share allocation by H-DRF on a simple example. Consider the hierarchy tree shown in Figure 8. We use 49 Amazon EC2 servers in this experiment, configuring hadoop to use 16GB memory and 4 CPUs and 4 GPUs on each server. The weights of parent nodes n_1:n_2 as 4:1. The weights for all other nodes are equal to 1. One long running job (2000 tasks) is submitted to each of the leaves in the hierarchy. Each task in the jobs submitted to $n_{1,1}$, $n_{1,2}$, $n_{2,1}$ and $n_{2,2}$ have

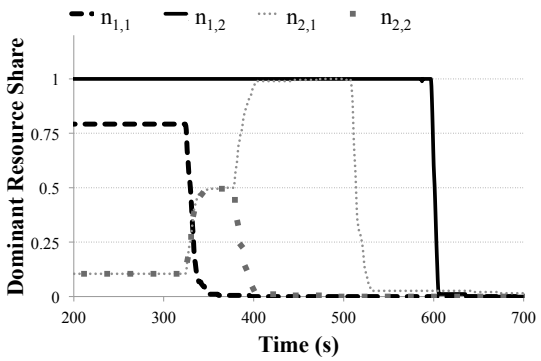


Figure 9: Resource sharing between the leaf nodes shown in Figure 8

resource requirements $\langle 1,1,0\rangle$, $\langle 1,0,1\rangle$, $\langle 1,1,0\rangle$ and $\langle 1,1,0\rangle$ respectively. Thus, the dominant resource of $n_{1,1}$, $n_{2,1}$ and $n_{2,2}$ is CPU, while the dominant resource of $n_{1,2}$ is GPU. Figure 9 shows the dominant share allocated by H-DRF to the various leaf nodes in the hierarchy across time. Between 200-300s all leaf nodes are active. The job submitted to $n_{1,1}$ ends at 350s, $n_{2,2}$ at 400s, $n_{2,1}$ at 550s and finally $n_{1,2}$ at 600s.

Weighted proportional fair sharing of dominant resources: The 4:1 ratio of weights between n_1 and n_2 requires that all children of n_1 combined should receive 0.8 share of the cluster. When all four leaf nodes in the hierarchy are active (between 200-300s), this is indeed the case. $n_{1,1}$ gets 0.8 share of the CPUs (its dominant resource) , while $n_{2,1}$ and $n_{2,2}$ receive a share of 0.1 each, making the total share of CPUs received by parent n_2 0.2 (*i.e.,* exactly $1/4$th that of n_1). Thus, H-DRF delivers proportional dominant resource sharing between all sibling nodes in this hierarchy. Also note that instead of $n_{1,2}$ receiving a 0.8 share of the GPUs in the cluster (its dominant resource), it receives a share of 1.0 because no other node in the hierarchy is contending for the cluster's GPUs. All the CPUs and GPUs in the cluster consumed by tasks demonstrates that H-DRF achieves a pareto-efficient[5] allocation.

Normalization in H-DRF: $n_{1,2}$'s dominant share of 1.0 does not affect the sharing between $n_{1,1}$, $n_{2,1}$ and $n_{2,2}$. H-DRF normalizes the share of $n_{1,2}$ to 0.8 to calculate the total allocation vector of n_1.

[5] pareto efficiency implies that no node in the hierarchy can be allocated an extra task on the cluster without reducing the share of some other node

Sharing of resources between sibling queues: Once $n_{1,1}$ completes at 300s, its resources are taken over by $n_{2,1}$ and $n_{2,2}$, taking their dominant resource (CPU) share to 0.5 each. The share of $n_{1,2}$ remains 1.0 because none of the other nodes require GPUs. This demonstrates the sibling sharing property where a node can increase its share to take advantage of a non-demanding sibling. This property is also exhibited when $n_{2,2}$ finishes at 400s, and $n_{2,1}$ increases its share to use all the CPUs in the cluster.

5.2 Comparison to existing Hadoop multi-resource schedulers

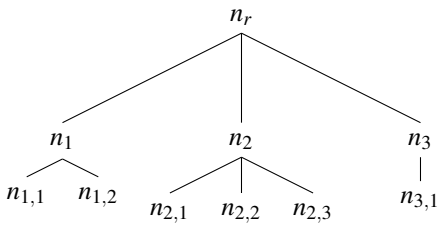


Figure 10: Hierarchy used in §5.2

In this section, we show that starvation does indeed occur when current open-source schedulers are used. We do so by running a workload and comparing the performance of the H-DRF prototype implementation to the Capacity scheduler implemented in Hadoop 2.0.2-alpha [6]. The Capacity scheduler performs hierarchical multi-resource scheduling in Hadoop in the manner described in §3.3. Our cluster consists of 50 Amazon EC2 servers, each configured to have 6 GB memory, 4 CPU cores and 1 GPU (Total 300GB, 200 cores and 50 GPUs). We run three schedulers - (i) an unchanged implementation of the Capacity Scheduler (henceforth named C.S-Current, and which is not pareto efficient), (ii) the pareto-efficient implementation of the same scheduler (Pareto-Efficient-C.S) and (iii) H-DRF, on the same workload and compare throughput and job response times.

Hierarchy Tree: The hierarchy tree chosen for this experiment is based on typical hierarchies seen by Cloudera, a cloud-based company with hundreds of customers running hierarchical scheduling, and is shown in Figure 10.

Input Workload: The input job schedule has the same job size distribution as a 10-day Facebook trace (collected in 2010). Table 1 shows the job sizes of the input job schedule in our experiment and the Facebook trace. We create a 100-job schedule by sampling job sizes[6] from the enterprise trace. If the sampled job has memory as its dominant resource in the Facebook trace, it is configured to use 1.5 GB memory per task, else it is configured to use 1 GB memory per task. All jobs request one CPU core and no GPUs per task, except the jobs submitted to $n_{1,2}$ which request one GPU core and no CPU. The 100 jobs are divided into six groups to be submitted to each of the leaf nodes in the hierarchy. The ten large jobs (>500 tasks) are divided between nodes $n_{1,1}$, $n_{1,2}$ and $n_{3,1}$, while the smaller jobs (<500 tasks) are divided between the leaf nodes of n_2. The order of the jobs submitted to the leaves is kept the same across the different experiments. Each leaf node runs its next job as soon as the previous one finishes. The jobs are written to precisely use the amount of memory and CPU specified.[7].

Comparison Metrics: We compare two metrics for the three schedulers. First we note the throughput of each leaf node in terms of the number of its tasks running at any point. Second, we calculate the improvement in job response time in H-DRF as compared to the other two schedulers. The percentage improvement in job response time for H-DRF is calculated as how much earlier the same job completed in H-DRF as compared to the other scheduler, as a percentage of the job duration time in the other scheduler.[8]

Table 1: Characteristics of the input job schedule. The input job schedule maintains the job size distribution from the Facebook trace

Bin (# tasks)	% Jobs in Facebook trace	Num. jobs in our workload
0-19	73	74
20-149	10	10
150-499	7	6
500+	9	10

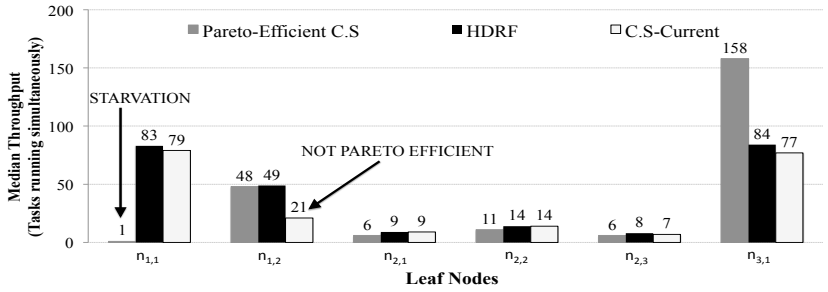
Comparison against C.S-Current: C.S-Current uses the naive H-DRF technique (§3.3) and tries to equalize the dominant share allocation between every pair of sibling nodes. However, it is not pareto-efficient because it stops allocating tasks to nodes as soon as any of the cluster resources is fully utilized. The consequence is that once all the CPUs in the cluster become utilized,

[6] number of tasks

[7] Note that due to some tasks requiring only 1GB of memory, there may be enough memory left over in each server after allocating four CPU-based tasks to allocate a GPU task

[8] Due to a higher throughput obtained by H-DRF in some cases, a particular job may start much earlier in H-DRF than in the other schedulers, leading to the percentage improvement in job response time to be more than 100%

(a) Throughput of leaf nodes under the current implementation of the Capacity Scheduler (C.S-Current), Capacity Scheduler modified to be Pareto-efficient(Pareto-Efficient-C.S), and H-DRF for the hierarchy tree in Figure 10.



(b) Percentage Improvement in Job Response Times on using H-DRF as compared to C.S-Current and Pareto-Efficient-C.S

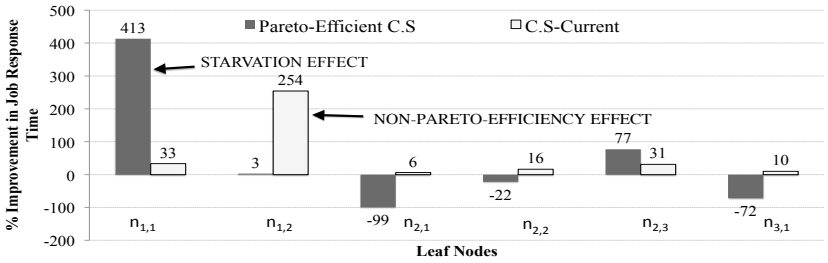


Figure 11

C.S-current stops scheduling tasks of $n_{1,2}$ (which uses GPUs) even though there are enough resources in the cluster for it to increase its share. Thus, $n_{1,2}$, whose dominant resource share is $21/50 = 0.42$, is pinned down to roughly the dominant resource share of $n_{1,1}$ (dominant share of $79/200 \approx 0.40$) (see Figure 11a). H-DRF, on the other hand, is Pareto-efficient and allows $n_{1,2}$ to increase its dominant share beyond that of $n_{1,1}$. The throughput of the remaining nodes are roughly equal for both schedulers. H-DRF improves its job response times for $n_{1,2}$ by almost 250% on the average (Figure 11b). [9]

Comparison against Pareto-Efficient-C.S: We then modified C.S-Current to enable pareto-efficiency by adding support for non-usage of an available resource, by removing task reservations on servers, and by continuing scheduling tasks if there exists leaf nodes that do not demand the saturated resources. On each server we maintain a headroom equal to the resources required by the largest task[10] to ensure that a task with large resource demands does not get starved out by tasks with smaller demands (the reason for task reservations in the C.S-Current). Figure 11a shows that in Pareto-Efficient-C.S (which is now exactly the naive H-DRF technique in §3.3), $n_{1,2}$'s share increases beyond its sibling's share to use almost all the GPUs in the cluster. However, the increase in $n_{1,2}$'s share also increases the dominant resource share of its parent node n_1 to 1.0. In an attempt to raise n_2 and n_3's dominant share to match that of n_1, Pareto-Efficient-C.S allocates any new CPU cores vacated by a finishing task of $n_{1,1}$ to a leaf node of n_2 or n_3. The starvation of $n_{1,1}$ gets so dire that its median throughput in Pareto-Efficient-C.S drops to just 1 task, with its fair share being divided among the remaining leaf nodes. The increased share for the leaf nodes of n_2 and n_3 leads to an improvement in their job response times over H-DRF. $n_{1,1}$ only achieves its fair share once all jobs of $n_{1,2}$ finish, resulting in H-DRF finishing jobs 413% faster.

5.3 Comparison to hierarchical slot-based Fairness: Prototype and Simulation Results

Slot-based schedulers divide a server's resources equally into a pre-defined number of slots, and assign one task to each slot with the objective of equalizing the number of tasks allocated to any pair of sibling nodes. As-

[9] Note that C.S-current behaves exactly like H-DRF and is pareto-efficient when every task uses at least some portion of every cluster resource.

[10] In this case $\langle 1.5, 1, 1 \rangle$

signing a task to a slot without enforcing that the task's resource requirements be lesser than the slot size, can lead to under or over-subscription of server resources depending on the slot count per server and the task sizes (as shown in [1]). Over-subscription of memory on a server may lead to thrashing which will dramatically reduce throughput and job response times. In section 5.3.1 we quantify the benefits of H-DRF over slot-based schedulers through our prototype implementation, and in section 5.3.2 through simulation.

5.3.1 Prototype results

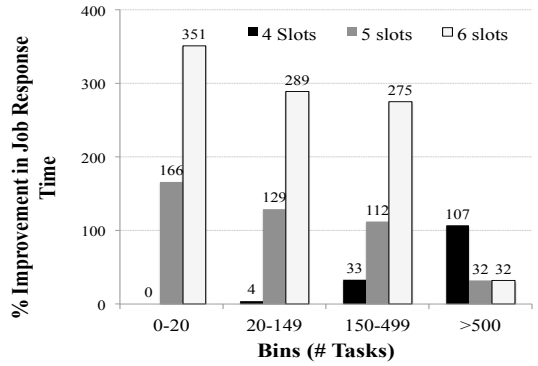


Figure 12: Improvement of average job response times by H-DRF against hierarchical slot-based scheduling on our implemented prototype

We use a 50-server Amazon EC2 cluster, each server having 7 GB memory and 8 CPUs. We configure each server to use 6 GB of memory (leaving 1 GB to be used by the Operating System). Each server is configured to behave as having 4 CPUs and 4 GPUs. We use the same job schedule and node hierarchy as the previous section (§5.2) and use the same definition of percentage improvement in job response time, except that we set the weight of n_3 to 2 in each run. We compare H-DRF against three possible configurations of the hierarchical slot scheduler - with 4, 5 and 6 slots per server.

The job schedule completes execution in 1379 , 1446, 1514 and 1732 seconds for H-DRF, 4-slot, 5-slot and 6-slot scheduling respectively. Figure 12 shows the percentage improvement in job response times obtained by H-DRF over slot scheduling. H-DRF can more efficiently pack jobs in a cluster, ensuring superior throughput and resulting in jobs finishing quicker. The 4-slot case gets worse for larger jobs because its throughput is lower than that of H-DRF. Since small jobs finish in a single wave, the increased throughput in H-DRF does not play a significant role in the comparison against 4-slots. From 5-slots onwards, small jobs (a majority of which use 1.5 GB memory for each of their tasks) encounter thrashing due to the slot scheduler overcommitting the amount of memory on the server. The 6-slot scheduler also encounters thrashing for smaller jobs, but by virtue of packing more tasks per server, its performance improves for larger jobs.

5.3.2 Simulation results

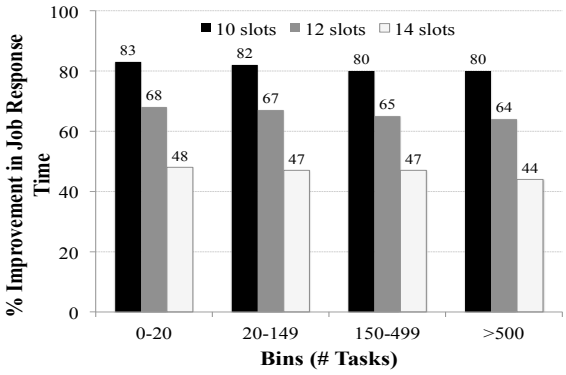


Figure 13: Improvement in job response times of H-DRF against hierarchical slot-based scheduling while simulating the Facebook trace

We also use a trace-driven simulator to compare H-DRF to slot scheduling. We use the same configuration as reported in [23]. The input to the simulator was a 10-day trace from a 2000-server cluster at Facebook. We simulate the trace on a 1200-node cluster in order to achieve high utilization and make fairness a relevant concern. The simulator re-creates the exact resource usage and time taken by each job in the Facebook trace.

In the trace, we found that memory requirements of each job could vary up to 9 GB, and CPU requirements up to 3 cores. Each server was configured to have 32 GB memory, 16 CPUs, in addition to which 200 of the servers had 16 GPUs. We use the same hierarchy as shown in Figure 10. Each leaf node was assumed to have 25 users submitting a mixed workload (small jobs and large jobs), with inter-arrival times sampled from the Facebook trace. The time of submission of a job was kept the same across all the experiments. If jobs got queued at a leaf node, they were served in a first-in-first-out manner. The simulation was stopped at 604800 seconds (or 1 week) , and the improvement in job response time achieved by H-DRF among completed jobs as compared to the slot scheduler with 10, 12 and 14 slots was computed. As shown in Figure 13, H-DRF improved the response times over hierarchical slot scheduling by 44-83% by achieving higher utilization.

6 Related Work

Our work builds on the notion of Dominant Resource Fairness [1]. DRF guarantees multi-resource fairness only for non-hierarchical systems, and has been extended to other non-hierarchical settings such as in [10, 11, 12, 13]. H-DRF extends the concept of Dominant Resource Fairness to the hierarchical setting, and provides new properties such as group-strategy proofness.

Hierarchical scheduling has been studied in many fields of Computer Science such as in networking [24] to allocate bandwidth between different classes of flows, in Operating Systems [25, 26] to support isolation and performance guarantees to different classes of applications. Ensuring storage performance requirements for different sub-organizations in a company arranged in a hierarchical fashion has been studied in [27]. Hierarchical scheduling has also been studied in grid computing for multi-grid resource allocation and management [28, 29]. Hierarchical schedulers such as Fair [9] and Capacity [6] have been implemented in Hadoop. The Hadoop Fair scheduler assigns resources at the slot granularity, which might lead to underutilization or oversubscription of resources on a server. H-DRF considers multiple resources during its allocation decisions, and hence avoids these issues.

Finally, the Hadoop Next-Generation (YARN) [19] has recently added multi-resource DRF support to its Capacity scheduler [18]. Furthermore, there is a recent JIRA from Cloudera on implementing a new DRF-based fair scheduler [20]. As we showed in our evaluation, the hierarchical implementation of DRF in YARN can leave resources unallocated and sometimes starve jobs. Our implementation of H-DRF in YARN does not suffer from these problems.

7 Conclusion and Future Work

Hierarchical scheduling is an essential policy that is supported by most cloud schedulers. Recently, multi-resource fairness has emerged as an important additional requirement for job scheduling to deal with heterogeneous workloads. For this reason, industry has developed two separate hierarchical schedulers for Hadoop YARN. Unfortunately, we show that they suffer from either job starvation or leave some resource unallocated despite demand. This is because multi-resource fairness introduces new challenges for hierarchical scheduling. We have proposed H-DRF, which is a hierarchical multi-resource scheduler for Hadoop. Our evaluation shows that it outperforms the traditional slot scheduling, and does not suffer from starvation and inefficiencies.

H-DRF presents several areas for future research. First, H-DRF does not deal with issues arising out of task placement constraints. The notion of dominant resource fairness under placement constraints is still an open question. Second, H-DRF's allocation vector update step may require recomputing the dominant shares of every other node in the hierarchy. This might be computationally expensive in a hierarchy with a large number of nodes. Third, pre-emptions could be added to H-DRF to enable nodes achieve their dominant shares faster.

8 Acknowledgments

We would like to thank Ganesh Ananthanarayanan and our shepherd Ajay Gulati for their valuable feedback. This research is supported in part by NSF CNS-1161813, NSF CNS-0931843, NSF CISE Expeditions award CCF-1139158, DARPA XData Award FA8750-12-2-0331, National Science Foundation under Grants CNS-0931843 (CPS-ActionWebs) and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware and Yahoo!

References

[1] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, I. Stoica, and S. Shenker. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[2] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[3] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[4] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. Static scheduling in clouds. In *HotCloud*, June 2011.

[5] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI*, December 2008.

[6] Hadoop Capacity Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[7] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters.

In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[8] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012.

[9] Hadoop Fair Scheduler. http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html.

[10] Carlee Joe-Wong, Soumya Sen, Tian Lan, and Mung Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, pages 1206–1214, 2012.

[11] Avital Gutman and Noam Nisan. Fair Allocation Without Trade. In *AAMAS*, June 2012.

[12] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities. In *ACM EC*, 2012.

[13] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *SIGCOMM*, 2012.

[14] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, October 2012.

[15] Bikash Sharma, Ramya Prabhakar, Seung-Hwan Lim, Mahmut T. Kandemir, and Chita R. Das. Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters. In *IEEE CLOUD*, pages 1–8, 2012.

[16] R. Boutaba, L. Cheng, and Q. Zhang. On cloud computational models and the heterogeneity challenge. *J. Internet Services and Applications*, 3(1):77–86, 2012.

[17] 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 *NSDI*, 2011.

[18] YARN DRF extension to the Capacity Scheduler. https://issues.apache.org/jira/browse/YARN-2.

[19] The Next Generation of Apache Hadoop MapReduce. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen.

[20] YARN DRF extension to the Fair Scheduler. https://issues.apache.org/jira/browse/YARN-326.

[21] Hadoop Yarn 2.0.2-alpha. http://hadoop.apache.org/docs/current/.

[22] Abhishek Chandra and Prashant Shenoy. Hierarchical scheduling for symmetric multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 19:418–431, 2008.

[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andrew Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. Technical Report UCB/EECS-2011-18, EECS Department, University of California, Berkeley, Mar 2011.

[24] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *IEEE/ACM Transactions on Networking*, pages 143–156, 1997.

[25] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *OSDI*, pages 107–121, 1996.

[26] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional Share Resource Management*. PhD thesis, MIT, Laboratory of Computer Science, September 1995. MIT/LCS/TR-667.

[27] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. Demand based hierarchical QoS using storage resource pools. In *Proceedings of the Annual USENIX Technical Conference*, 2012.

[28] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Grid ComputingGRID 2000*, pages 191–202. Springer, 2000.

[29] V. Subramani, R. Kettimuthu, S. Srinivasan, and S. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 359 – 366, 2002.