Fair Resource Allocation for Data-Intensive Computing in the Cloud

Shanjiang Tang, Bu-Sung Lee, and Bingsheng He

Abstract—To address the computing challenge of 'big data', a number of data-intensive computing frameworks (e.g., MapReduce, Dryad, Storm and Spark) have emerged and become popular. YARN is a de facto resource management platform that enables these frameworks running together in a shared system. However, we observe that, in cloud computing environment, the fair resource allocation policy implemented in YARN is *not* suitable because of its *memoryless* resource allocation fashion leading to violations of a number of good properties in shared computing systems. This paper attempts to address these problems for YARN. Both single-level and hierarchical resource allocations are considered. For single-level resource allocation, we propose a novel fair resource allocation mechanism called *Long-Term Resource Fairness (LTRF)* for such computing. For hierarchical resource allocation, we propose Hierarchical Long-Term Resource Fairness (H-LTRF) by extending LTRF. We show that both LTRF and H-LTRF can address these fairness problems of current resource allocation policy and are thus suitable for cloud computing. Finally, we have developed LTYARN by implementing LTRF and H-LTRF in YARN, and our experiments show that it leads to a better resource fairness than existing fair schedulers of YARN.

Index Terms—MapReduce, hadoop, fair scheduler, YARN, cloud computing, long-term resource fairness

✦

1 INTRODUCTION

NOWADAYS, we have entered the era of 'big data', where the data is collected at unprecedented scale in many application areas, including e-commerce [1], social network [18], and computational biology [43]. Large-scale data processing is thus needed in analyzing and mining of massive data. In recent years, a number of large-scale data-intensive computing frameworks (e.g., MapReduce [2], Spark [13], Dryad [6], HIVE [25]) have been developed for different applications/data (e.g., batch/iterative applications, graph/streaming data). Recently, YARN [3] has emerged as a popular distributed resource management system that enables a number of these data-intensive computing frameworks (e.g., MapReduce [2], Spark [13], HIVE [25]) to efficiently share a cluster.

Moreover, cloud computing has emerged as a popular platform for users to compute their large-scale data applications, attracting from its merits such as flexibility, elasticity, and cost efficiency. Resource utilization is a key design issue in the cloud for both users and providers [47]. However, the fact is that the resource utilization of current data-intensive computing systems is far from ideal. Delimitrou and Kozyrakis [48] had an analysis of Twitter production cluster over one month. However, they showed that the majority of servers (e.g., 80 percent servers) are below 20 percent utilization.

Resource sharing is a classical and effective approach for high resource utilization [23]. It is based on the observations

that 1). different users often have different resource demands; 2). even for an individual user, her demand is changing over time. Resource sharing can thereby achieve a better utilization than the non-sharing case by enabling overloaded users to utilize unused resources from underloaded users. In the cloud environment, we can establish a multi-tenant computing system by importing all computing instances rented by each user. The computing resources of the system can be managed and shared between users in their analytical data computation with existing resource management systems such as YARN [3].

Fairness is an important system issue in resource sharing. Only when the fairness is guaranteed for users, the resource sharing can be possible among different users. One of the most popular fair allocation policy is *(weighted) max-min fairness* [23], which maximizes the minimum resource allocation obtained by a user in a shared computing system. It has been used in YARN [3] as well as other popular resource sharing systems such as Mesos [10]. Unfortunately, we observe that the fair polices implemented in these systems are *memoryless*, i.e., allocating resources fairly at *instant* time without considering history information. We refer these policies with *MemoryLess Resource Fairness (MLRF)*. MLRF is *not* suitable for cloud computing system due to the following problems:

Cost-inefficient workload submission problem. For *MLRF*, there is an implicit assumption that all users are unselfish and honest towards their requested resource demands, which is however often not true in practice. It can cause cost-inefficient workload submission problem with *MLRF* in the sharing environment. Consider two users A and B sharing a system for example. Let D_A and D_B be the *true* workload demand for A and B at time t_0, respectively. Assume that D_A is less than its share[1] while D_B is larger than its share. When it is in the

- *S.J. Tang is with the School of Computer Science and Technology, Tianjin University, Tianjin, China. E-mail: tashj@tju.edu.cn.*
- *B.S. Lee and B.S. He are with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. E-mail: {ebslee, bshe}@ntu.edu.sg.*

1. By default, we refer to the *current* share at the designated time (e.g., t_0), rather than the *total* share accumulated over time.

exclusively non-sharing computing environment, there is no incentive/benefit for user A to run dirty (i.e., cost-inefficient) tasks for possessing all her unused resources since there is no resource contention for her in this case at any time. However, when it comes to the sharing case, if A yields her unused resources to B at t_0, next at time t_1 when A has many tasks to compute, it would make her be pending/waiting for possessed resources to be released from B. Assuming A is a selfish user, in order to avoid that in the sharing case, it is feasible for A to possess all of her unused resources by submitting a number of small-size dirty (i.e., cost-inefficient) tasks at t_0 until the true resource demand of A become larger than or equal to her share in next computation stages. If all underloaded users in the sharing system behave like A, it then causes the cost-inefficient problem for running workloads and breaking the sharing incentive (See definition in Section 3) property at the same time.

Strategy-proofness problem. In a fairly shared system, it is important to ensure that no users can get any benefits by lying (See *Strategy-proofness* in Section 3). We argue that *MLRF* cannot satisfy this property. Consider a Hadoop system consisting of three users A, B, and C. Assume A and C are honest users while B not. It could happen at a time that both the *true* map slots demands of A and B are less than their own shares while C's *true* map slots demand exceeds its share. In that case, A yields her unused map slots to others honestly. But B can cheat the Hadoop system by *falsely* reporting her map slots demand (say, e.g., far larger than her share) and put some computing load of her reduce tasks in the map tasks, which can be achieved by modifying the application code [23]. Then B can compete with C for unused map slots from A, benefiting B for having more workloads computed and hence violating strategy-proofness. Moreover, it will break the sharing incentive property as well if all other users do the same thing like her.

Resource-as-you-contributed unfairness problem. In the shared cloud system, we should ensure that the total resources received by each user are proportional to her resource contribution (See *resource-as-you-contributed Fairness* in Section 3). Due to the varied resource demands (e.g., workflows) for a user at different time, *MLRF* fails at this point. Consider two users A and B. At time t_0, it could happen that the demand D_A is less than its share and hence its extra unused resource will be possessed by user B (i.e., lend to B) according to the work conserving property of *MLRF*. Next at time t_1, assume that user A's demand D_A becomes larger than its share. With *MLRF*, however, A can only use her current share (i.e., cannot get lent resources at t_0 back from B), if D_B is larger than its share, due to the *memoryless* feature. It is unfair for A to get the amount of resources that she should have obtained from a long-term view.

In this paper, we propose *Long-Term Resource Fairness (LTRF)* and show that it can solve the aforementioned problems. We start with the single-level resource allocation, and next extend it to hierarchical resource allocation [24]. For the single-level resource allocation, we demonstrate that LRTF has good properties that are important for fair resource allocation on the shared cloud system. Five such properties are sharing incentive, cost-efficient workload incentive, resource-as-you-contributed fairness, strategy-proofness and Pareto Efficiency. *LTRF* provides incentives for users to submit meaningful workloads and share resources by ensuring that no user is better off in the exclusively non-sharing computing system than in the sharing case. Moreover, *LTRF* can guarantee the amount of resources a user should receive in terms of the amount of resources she contributed, in the case that her resource demand varies over time. In addition, *LTRF* is strategy-proof, as it can make sure that a user cannot get more resources by lying about her resource demand.

We have extended LTRF to support hierarchical resource allocation by considering the organizational priorities in resource allocations. We show that the combination of hierarchical and long-term resource allocation brings new challenges that do not exist in the single-level long-term resource allocation. A naive extension of LTRF can lead to the starvation. To solve this problem, we propose a starvation-aware Hierarchical LTRF (H-LTRF) based on the *timeout* technique.

We have implemented LTRF and H-LTRF in YARN [3], by developing a long-term fair scheduler *LTYARN*. The experiments show that, 1). LTRF can guarantee Service-Level Agreement(SLA) via minimizing the sharing loss and bringing much sharing benefit for each user (See Fairness definition in Section 4.2), whereas MLRF not; 2). the shared methods using either LTRF and MLRF can possibly get better performance than non-shared one, or at least as fast in the shared system as they do in the non-shared partitioning case. The performance finding is consistent with previous work such as Mesos [10]; 3). H-LTRF can address the possible starvation problem in hierarchical resource allocation.

This paper is organized as follows. Section 2 gives the background and motivation. Section 3 presents several cloud-oriented resource allocation properties. Section 4 models the problem for single-resource allocation and defines the notion of fairness for cloud computing. Section 5 gives the design and principle of fair policies. Section 6 presents the implementation details of LTYARN. Section 7 evaluates the fairness and performance of LTYARN experimentally. Section 8 reviews the related work. Finally, we conclude the paper in Section 9.

2 BACKGROUND AND MOTIVATION

In this section, we start with the background of data intensive computing in the cloud and present our work settings. Next, we give an introduction of resource allocation in YARN.

2.1 Data-Intensive Computation in the Cloud

To enable the query and analysis for such a large volume of data, a number of large-scale data management and parallel computing systems have been designed and built, including Hadoop [15], Hive [25], Dryad [6], and Spark [13]. On the other hand, there is a trend for users to take cloud computing as a computing infrastructure for data-intensive computing due to its capacity of elastic computing and storage resources [46], and pay-as-you-go pattern for rented cloud resources.

Running in the cloud, applications need to have a high utilization (and in turn the high cost efficiency) for rented cloud resources. Otherwise, the cloud resources are idle, resulting in a waste of money as well as resources [5]. In practice, it is most likely that the resource demand of a

single user's application is changing over time, implying that it is hard to keep the high resource utilization all the time. More effective resource sharing is critical for improving the resource utilization in the cloud.

In this paper, let's consider a cloud-based computing system shared by n users, where user i has a resource contribution of k_i to the pool of cloud resources. To enable resource sharing sustainable between users in the long run, we should guarantee the proportional relationship between the amount of total resources a user used over a period of time and the amount of resources contributed by the user to the shared cloud (i.e., resource-as-you-contributed fairness). Our aim thus turns to explore a fair resource allocation policy that can meet all the aforementioned *good* properties listed in Section 3.

2.2 Resource Management in YARN

YARN, as the next generation of Hadoop (i.e., Hadoop MRv2), has evolved to be a large-scale data operating platform and cluster resource management system. There is a new architecture for YARN, which separates the resource management from the computation model. Such a separation enables YARN to support a number of diverse data-intensive computing frameworks including Dryad [6], Giraph, Hoya, Spark [13], Storm [12] and Tez. In YARN's architecture, there is a global master named *ResourceManager (RM)* and a set of per-node slaves called *NodeManagers(NM)*, which forms a generic system for managing applications in a distributed manner. The RM is responsible for tracking and arbitrating resources among applications. In contrast, the NM has responsibility for launching tasks and monitoring the resource usage per slave node. Moreover, there is another component called *ApplicationMaster(AM)*, which is a framework-specific entity. It is responsible for negotiating resources from the RM and working with the NM to execute and monitor the progress of tasks. Particularly, all resources of YARN are requested in the form of '*container*', which is a logical bundle of resources (e.g., <1 CPUs, 2G memory>).

As a multi-tenant platform, YARN organizes users' submitted applications into queues and share resources between these queues. Users can set their own queues in a configuration file provided by YARN. When all users' queues are configured at the same level, the cluster resources will then be allocated at one level, which we call the *single-level* resource allocation. Moreover, to reflect the hierarchical tree structure for organizations of users in practice, YARN also supports hierarchical queues of tree topology. Each queue can represent an organization or a user. In the tree topology, there is a root node called *Root Queue*. It distributes the resources of the whole system to the intermediate nodes called *Parent Queues*. Each parent queue further re-distributes resources into its sub-queues (parent queues or leaf queues) recursively until to the bottom nodes called *Leaf Queues*. Finally, users' submitted applications within the same leaf queue share the resources. We call this allocation as *hierarchical* resource allocation.

There is a Fair Scheduler [22] inside YARN, which can support both single-level and hierarchical resource allocations. Moreover, both single-resource and multi-resource allocations are also supported. For the single-resource allocation, current version of YARN adopts the max-min fair policy and focuses only on the memory resources. With regard to the multi-resource allocation, it takes the Dominant Resource Fairness (DRF) [23] and considers both CPU and memory resources.

In our paper, we focus on the single-resource allocation for YARN by considering both single-level and hierarchical resource allocation in cloud computing. We remain the consideration for multi-resource allocation as future work.

3 CLOUD-ORIENTED RESOURCE ALLOCATION PROPERTIES

We present a set of desirable properties for cloud computing. Based on these properties, we design our fair allocation policies for YARN. We have found the following five important properties:

- *Sharing incentive*: Each user should be better off sharing the resources with others, than exclusively using the resources individually. Consider a cloud system equally shared by n users over t period time. Then each user should get at least $t \cdot \frac{1}{n}$ resources in the shared system.

- *Cost-efficient workload incentive*: Resources in the cloud are priced (i.e., not free). In a shared cloud system, we should encourage users to submit workloads that generate positive utility to them (i.e., cost-efficient workload) for cost efficiency and avoid those spam workloads with no positive utility (i.e., cost-inefficient workload). That is, a user should be better off submitting cost-efficient workload and yielding unused resources to others when not needed. Otherwise, she may be selfish and possesses all unneeded resources under her share by submitting cost-inefficient tasks in a shared computing environment.

- *Resource-as-you-contributed fairness*: In the cloud, assume that each user contributes a certain number of machines (resources) to its common pool of machines (resources). Then, the accumulated resource that each user *used* received over time should be in proportion to her contribution in the shared environment. This property is important as it is a Service-Level Agreement guarantee for users.

- *Strategy-proofness*: Users should not be able to get benefits by lying about their resource demands. This property is compatible with sharing incentive and resource-as-you-contributed fairness, since no user can obtain more resources by lying.

- *Pareto efficiency*: In a shared resource environment, it is impossible for a user to get more resources without decreasing the resource of at least one user. This property can ensure the system resource utilization to be maximized.

Although this paper is focused on YARN, it is worth mentioning that our methodology can be applied to other resource management systems such as Mesos [10].

4 SYSTEM MODEL AND FAIRNESS DEFINITION

In this section, we first model the single-resource allocation for YARN in cloud environment. Next, we give the definition
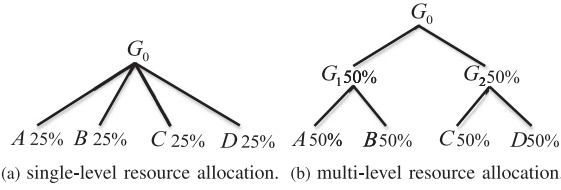
(a) single-level resource allocation. (b) multi-level resource allocation.

Fig. 1. A comparison example of the long-term resource fairness between single-level and multi-level resource allocations for four users A, B, C and D with equal share of the whole resource service, where G_i denotes the ith group and G_0 represents the total resource service of the whole system.

of fairness, which is used to assess fair policies in the following sections.

4.1 System Model

This paper considers the single-resource allocation fairness (e.g., memory) for YARN in the most commonly used *homogeneous* environment [3]. Let $M = \{1 \ldots m\}$ be the set of machines (or instances) in the shared computing system. For each machine, we assume the amount of resources (e.g., memory) is R_i. Thus, the total resource capacity R of the system is $R = \sum_{i=1}^{m} R_i$. Let $N = \{1 \ldots n\}$ be the set of users in the shared computing systems. Assume that the resource contributions (i.e., shared weights) for n users are $W = \{w_1 \ldots w_n\}$. According to the practical needs of resource allocation, these users can be grouped into multiple queues of either single-level resource allocation structure or hierarchical resource allocation structure in YARN.

Without loss of generality, for example, there is a cloud system consisting of 100 instances of t2.medium type on the cloud, contributed by four users A, B, C and D with diverse data-intensive workloads (e.g., MapReduce, Tez, HIVE, and Spark) equally. In that case, we can establish a shared computing system with YARN. According to practical needs, the four users can be organized either into a single group for single-level resource allocation (e.g., Fig. 1a) or multiple groups for hierarchical resource allocation (e.g., Fig. 1b).

In our work below, we focus on the fairness for these two types of resource allocation structures for n users, namely, single-level resource allocation (Section 5.1) and hierarchical resource allocation (Section 5.2).

4.2 Fairness Definition

We consider the fairness from the resource allocation perspective. In a shared cloud environment, ideally, every user wants to get more resources or at least the same amount of resources in a shared computing system than the ones of exclusively using her partition of the system. We call it *fair* for a user (i.e., sharing benefit) when that can be achieved. In contrast, due to the resource contention in the shared system, it is also possible for the total resources a user received are less than that without sharing, which we call *unfair* (i.e., sharing loss). To ensure resource-as-you-contributed fairness and the maximization of sharing incentive property in the shared system, it is important to minimize *sharing loss* firstly and then maximize *sharing benefit*.

In the remainder of this paper, we refer to the *total* resources as *accumulated* resources along the time. Let $g_i(t)$ be the currently allocated resources for the ith user at time t.

Let $f_i(t)$ denote the *accumulated* resources for the ith user at time t. Thus,

$$f_i(t) = \int_0^t g_i(t)\, dt. \tag{1}$$

Let $d_i(t)$ and $S_i(t)$ denote the current demand and current resource share for the ith user at time t, respectively. Given the total resource capacity R of the system and the shared weight w_i for the ith user, there is

$$S_i(t) = R \cdot w_i \Big/ \sum_{k=1}^{n} w_k. \tag{2}$$

The fairness degree $\rho_i(t)$ for the ith user at time t is defined as the normalization result of the amount of resources a user obtained in a shared environment with respect to the non-shared environment, i.e.,

$$\rho_i(t) = \frac{AllocationResultWithSharing}{AllocationResultWithoutSharing} = \frac{\int_0^t g_i(t)\, \mathbf{d}t}{\int_0^t \min\{d_i(t), S_i(t)\}\mathbf{d}t}. \tag{3}$$

$\rho_i(t) \geq 1$ implies the absolute resource fairness for the ith user at time t. In contrast, $\rho_i(t) < 1$ indicates *unfair*. We can easily see that for a user i in a non-shared partition of the system, it always holds $\rho_i(t) = 1$, since it has $g_i(t) = \min\{d_i(t), S_i(t)\}$ at any time t in this scenario. To measure how much better or worse for sharing with a fair policy than without sharing (i.e., $\rho_i(t) - 1$), we propose two concepts *sharing benefit degree* and *sharing loss degree* to quantify it, respectively. Let $\Psi(t)$ be *sharing benefit degree*, as a sum of all $(\rho_i(t) - 1)$ subject to $\rho_i(t) \geq 1$, i.e.,

$$\Psi(t) = \sum_{i=1}^{n} \max\{\rho_i(t) - 1, 0\}, \tag{4}$$

and let $\Omega(t)$ denote *sharing loss degree*, as a sum of all $(\rho_i(t) - 1)$ subject to $\rho_i(t) < 1$, i.e.,

$$\Omega(t) = \sum_{i=1}^{n} \min\{\rho_i(t) - 1, 0\}. \tag{5}$$

Thereby, it always holds that $\Psi(t) \geq 0 \geq \Omega(t)$. Moreover, we see that in a non-shared partition of the computing system, it always holds $\Psi(t) = \Omega(t) = 0$, indicating that there are neither sharing benefit nor sharing loss. In contrast, in a shared cloud computing system, either of them could be nonzero. For a good fair policy, it should be able to maximize $\Omega(t)$ first (e.g., $\Omega(t) \rightarrow 0$) and next try to maximize $\Psi(t)$ as much as possible. Finally, we can use this two metrics to compare the fairness among different policies.

5 FAIR POLICY DESIGN AND PRINCIPLE FOR YARN

In this section, we give our design and principle of fair policies for YARN under cloud computing environment. Both single-level resource allocation and hierarchical resource allocation are considered.

5.1 Single-Level Resource Allocation

For single-level resource allocation, we first give a motivation example to show that MemoryLess Resource Fairness is *not* suitable for cloud computing system. Then we

TABLE 1
A Comparison Example of *MemoryLess Resource Fairness* and *Long-Term Resource Fairness*
in a Shared Computing System Consisting of 100 Computing Resources for Two Users A and B

	User A					User B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	−30	100	100	80	80	+30
t_2	40	40	40	60	−10	60	80	60	140	+10
t_3	80	80	50	110	0	50	70	50	190	0
t_4	60	90	50	**160**	0	50	70	50	**240**	0

(a) *Allocation results based on* MLRF. *Total Demand refers to the sum of the new demand and accumulated remaining demand in previous time.*

	User A					User B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	−30	100	100	80	80	+30
t_2	40	40	40	60	−10	60	80	60	140	+10
t_3	80	80	80	140	+30	50	70	20	160	−30
t_4	60	60	60	**200**	+10	50	100	40	**200**	−10

(b) *Allocation results based on* LTRF.

propose Long-Term Resource Fairness, a cloud-oriented allocation policy to address it and meet the desired properties in the previous section.

Motivation example. Consider a shared computing system consisting of 100 resources (e.g., 100 GB RAM) and two users A and B with equal share of 50 GB each. Without loss of generality, let's consider a simple case for the ease of clarification by assuming that the resource demand and execution time of all tasks are 1 GB and 1 time unit (e.g., 1 min), respectively. But it is worthy mentioning that in our real implementation of LTRF (Section 6.2), we do not have any assumption about the execution time and submission time of tasks. As illustrated in Table 1, assume that the new requested demands at time t_1, t_2, t_3, t_4 for user A are $20, 40, 80, 60$, and for user B are $100, 60, 50, 50$, respectively. With MLRF, we see in Table 1(a) that, at t_1, the total demand and allocation for A are both 20. It lends 30 unused resources to B and thus 80 allocations for B. The scenario is similar at t_2. Next at t_3 and t_4, the total demand for A becomes 80 and 90, bigger than its share of 50. However, it can only get 50 allocations based on MLRF, being *unfair* for A since it makes the total allocations for A and B become $160 (= 20 + 40 + 50 + 50)$ and $240 (= 80 + 60 + 50 + 50)$ at time t_4, respectively. Instead, if we adopt LTRF, as shown in Table 1(b), the total allocations for A and B at t_4 will finally be the same (e.g., 200), being *fair* for A and B.

LTRF scheduling algorithm. Algorithm 1 shows pseudocode for LTRF scheduling. It considers the fairness of the total amount of allocated resource consumed by each user, instead of currently allocated resources. The core idea is based on the '*loan(lending) agreement*' [7] with free interest. That is, a user will yield her unused resources to others as a *lend* manner at a time. When she needs at a later time, she should get the resources back from others that she yielded before (i.e., *return* manner). In our previous two-user example with LTRF in Table 1(b), user A first lends her unused resources of 30, 10 to user B at time t_1 and t_2, respectively. However, at t_3 and t_4, she has a large demand and then

collects all 40 extra resources back from B that she lent before, making *fair* between A and B.

Algorithm 1. LTRF Pseudo-Code

1: R**:** total resources available in the system.
2: $\ddot{R} = (\ddot{R}_1, \ldots, \ddot{R}_n)$**:** currently allocated resources. $\ddot{R}_i$ denotes the currently allocated resources for user i.
3: $U = (u_1, \ldots, u_n)$**:** total used resources, initially 0. u_i denotes the total resource consumed by user i.
4: $W = (w_1, \ldots, w_n)$**:** weighted share. w_i denotes the weight for user i.
5: **while** there are pending tasks **do**
6:     **Choose** user i with the smallest total weighted resources of u_i/w_i.
7:     $d_i \leftarrow$ the next task resource demand for user i.
8:     **if** $\ddot{R} + d_i \leq R$ **then**
9:         $\ddot{R}_i \leftarrow \ddot{R}_i + d_i$.   ▷ Update currently allocated resources.
10:        Update the total resource usage u_i for user i.
11:        Allocate resource to user i.
12:    **else**   ▷ The system is fully utilized.
13:        **Wait** until there is a released resource r_i from user i.
14:        $\ddot{R}_i \leftarrow \ddot{R}_i - r_i$. ▷ Update currently allocated resources

---

Finally, we make a property analysis in Appendix B, which can be found on the Computer Society Digital Library at http://doi.10.1109/TSC.2016.2531698, showing that LTRF satisfies all the desired properties in Section 3.

5.2 Hierarchical Resource Allocation

In previous sections, we have considered the single-level (i.e., user-level) resource allocation for LTRF. It allocates the unused resources of a user always to the person with lowest total consumed resources among all users. For example, in Fig. 1a , there are four users A, B, C and D with equal shares (i.e., 25 percent) of the whole cluster under the single-level resource allocation. When User A has unused resources, it will always yield to the user with lowest total resources

among B, C and D. However, if A want to lend her unused resources to B before considering C and D, and vice versa (i.e., *resource lending affinity*[2] between A and B), the single-level resource allocation cannot help. Instead, the multi-level resource allocation achieve the affinity requirement due to its recursively collective resource allocation feature [24].

Assume that there are two resource lending affinities between A and B, C and D, respectively. To satisfy such affinities, we create a hierarchy by grouping A and B, C and D to form a tree structure, as illustrated in Fig. 1b , where the internal nodes denote *groups* and the leaf nodes represent individual users that are ultimately to be allocated resources.

5.2.1 A Naive Approach

The hierarchical multi-level resource allocation follows a up-to-down collective resource allocation process [24]. We can extend the long-term fairness to multi-level resource allocation in the following way.

Given a set of users with demands and a set of resources, we record the total amount of resources assigned to each leaf node (user) in the hierarchy over time. Non-leaf nodes (i.e., groups) are simply assigned the sum of all total resources assigned to their immediate children. Under these definitions, the allocation starts at the root of the tree, and traverses down by picking the *demanding* child (i.e., the node with pending tasks) that has the lowest total amount of allocated resources, and allocates resources to it. When there is a tie, we randomly choose a node to allocate.

For example, the allocation for the hierarchy in Fig. 1b can be performed as follows. Whenever there are idle resources for the root node G_0, it will choose the node with the lowest total amount of allocated resources among its children G_1 and G_2. Assume that at the moment, total resources allocated for users A, B, C and D are $200, 10, 50$ and 60, respectively. Then the total amount of allocated resources for G_1 and G_2 are 210 and 110, respectively. In that case, G_2 is picked. Thereafter, the allocation continues between the immediate children C and D of G_2. According to LTRF, the leaf-node C is finally allocated and its total amount of allocated resources will be added up accordingly.

Starvation. We show that the naive approach can lead to starvation problem for users with the lowest total amount of allocated resource due to the collective resource allocation. Let us consider an example with the hierarchy given in Fig. 1b. Assume that there are 12 resource units for the system and four demanding leaf nodes (users) A, B, C and D at the beginning with the resource demands of $200, 10, 50$ and 60, respectively. For simplicity, let's assume that each task consumes one resource unit and lasts one time unit (e.g., 1 min).

As the allocation results illustrated in Fig. 2, the algorithm starts by allocating 3 resource units to each of A, B, C and D until the 3rd time unit. At the 3th time unit, B only needs 1 resource unit. A, C and D get $5, 3$ and 3 resource units, respectively. From the 4th to 16th time unit, the allocations occur among A, C and D with $6, 3$ and 3, respectively. At the 16th time unit, C needs only two resource units and the allocation will be $6, 2$ and 4 for A, C and D,

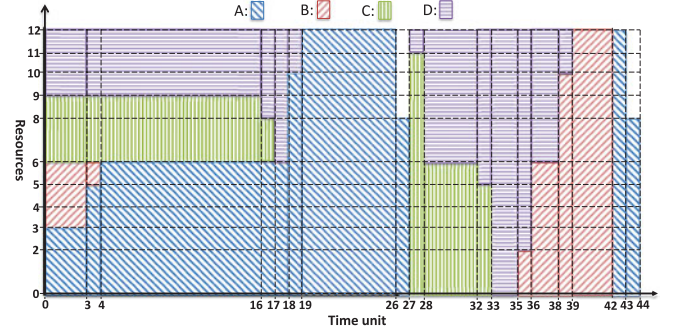2. Affinity users have higher priority to get unused resources than non-affinity users.

Fig. 2. The resource allocation results with the naive approach for four users A, B, C and D in the hierarchy of Fig. 1b . At the beginning, the resource demands for A, B, C and D are $200, 10, 50$ and 60, respectively. Next, the new demands arrive at time 27th time unit, that are $20, 60, 40$ and 80 for A, B, C and D, respectively.

respectively. Thereafter, the allocation will happen between A, D each with the allocated resources of 6 until D completes all tasks. After that, there is only one demanding user A and the whole resources will be given to A until the 27th time unit. At the 27th time unit, all tasks have been allocated. The total amount of allocated resources for users A, B, C and D are $200, 10, 50$ and 60, respectively. In that case, the total amount of allocated resources for G_1 and G_2 are $210, 110$, respectively. At 28th time unit, let's assume that there are new resource demands of $20, 60, 40$ and 80 for leaf nodes A, B, C and D, respectively. Although leaf node B is the user with the lowest total amount of allocated resource, it will be starved for a long time (e.g., at least $\lfloor (210 - 110)/12 \rfloor = 8$ minutes) before being allocated due to its sibling node A who has consumed too many resources and in turn makes G_1 much larger than G_2 (i.e., $210 - 110 = 100$ resource units).

5.2.2 Starvation-Aware Hierarchical LTRF (H-LTRF)

We propose a starvation-aware Hierarchical LTRF algorithm, namely H-LTRF. The basic idea is that, instead of strictly following LTRF for internal nodes in each resource allocation phase, we can relax such a constraint by allowing the internal nodes on the path containing the starved user to have a higher priority to get resource allocated. Such a modification might make the resource allocation violate the LTRF among some sibling internal nodes, possibly making the difference of the total amount of allocated resources between sibling internal nodes temporally become larger. However, since LTRF considers the total amount of allocated resources over time, the difference is recorded and LTRF can minimize such a difference in later resource allocation according to its *lending agreement* mentioned in Section 5.1.

For example, in Fig. 1b , when the leaf node B is recognized to be a starvation user, the internal node G_1 on its path $G_0 \rightarrow G_1 \rightarrow B$ will be given a higher priority in resource allocation than G_2, no matter the total amount of allocated resources of G_1 is larger than G_2 or not. Such allocation can address the starvation problem for B but enlarges the difference of the total amount of allocated resources between G_1 (i.e., 211) and G_2 (i.e., 110) at the moment. The difference is recorded and in later allocation, LTRF will help minimize it.

Algorithm 2 shows the pseudo-code for our starvation-aware hierarchical LTRF resource allocation. It consists of

two phases, i.e., *starvation detection phase* and *dynamic resource allocation phase*.

The starvation detection is based on a time-out technique. We provide a waiting time threshold T_{wait}. When a demanding (i.e., with pending tasks) leaf node, 1) has the lowest total amount of allocated resources among all demanding users; 2) has waited for a longer time than T_{wait} without being allocated since its last time allocation, it is assumed to be a starvation node. In the dynamic resource allocation phase, if the starvation node is detected, the resources will be allocated to it. Otherwise, it keeps the same hierarchical collective resource allocation approach as the naive approach.

Algorithm 2. H-LTRF Pseudo-Code

1: $G = \langle G_0, G_1, \ldots, G_i, \ldots \rangle$: Group node. G_i denotes the ith group. G_0 represents the whole cluster.
2: $S(G_i)$: the set of subgroups of G_i. *NULL* when G_i is a leaf group node (i.e., user node).
3: $U(G_i)$: the total amount of used resources for Group G_i, initially 0.
4: $\ell(G)$: the set of leaf group nodes (i.e., user nodes).
5: $G_i.wait$: the waiting time for Group G_i, initially 0.
6: T_{wait} : the maximum waiting time threshold.
7: **while** there are idle resources **do**
8:    $starvedNode = STARVATIONDETECTION(\ell(G))$;
                                 ▷ Starvation detection.
9:    **if** $starvedNode$ is not **NULL then**   ▷ Find starved node.
10:       Allocate resources to the leaf node $starvedNode$.
11:    **else**   ▷ No Starvation.
12:       $HIERARCHALLONGTERMALLOC(G_0)$.
13: **function** STARVATIONDETECTION(GroupSet G')
14:    **for** G_i in G' **do**   ▷ Update waiting time.
15:       **if** G_i has no pending tasks **then**
16:          $G_i.wait = 0$.
17:    G_i = demanding group with lowest total used resource in G'.
18:    **if** $G_i.wait > T_{wait}$ **then**   ▷ T_{wait}: waiting time threshold.
19:       $G_i.wait = 0$.
20:       **return** G_i.
21:    **else**
22:       **return** NULL.
23: **function** HIERARCHALLONGTERMALLOC(Group G_i)
24:    **if** G_i is not a leaf group node (user) **then**
25:       G_j = group node with lowest total used resource in $S(G_i)$ and pending tasks.
26:       $HIERARCHALLONGTERMALLOC(G_j)$.
27:    **else**   ▷ G_i is a leaf group node (user).
28:       Allocate resources to the leaf node G_i and set $G_i.wait = 0$
29:       Update $U(G_i)$ and set $G_i.wait = 0$.

By incorporating the timeout (i.e., T_{wait}) technique, we propose a starvation-aware H-LTRF that can address the starvation problem of the naive approach. Interestingly, we observe that the proposed starvation-aware H-LTRF is a generalization of the single-level LTRF and the naive approach, given by Theorem 1.

Theorem 1. *When $T_{wait} = 0$, H-LTRF allocates resources the same as the single-level LTRF. In contrast, when $T_{wait} = +\infty$, H-LTRF is the same as the naive approach.*

TABLE 2
List of Properties for MLRF, LTRF and H-LTRF

Property	Allocation Policy		
	MLRF	LTRF	H-LTRF
Sharing Incentive	✓	✓	✓
Cost-Efficient Workload Incentive		✓	✓
resource-as-you-contributed Fairness		✓	✓
Strategy-Proofness		✓	✓
Pareto Efficiency	✓	✓	✓

The proof Theorem 1 is given in Appendix A of the supplemental material, available online. It implies that there is a tradeoff between the resource lending affinity and starvation minimization. That is, when T_{wait} is small, it favors the starvation minimization. In contrast, when T_{wait} is large, it is beneficial to the resource lending affinity. The administrator can balance such a tradeoff by configuring the T_{wait} flexibly. Lastly, the guidance on how to configure T_{wait} is given in Section 7.6.

Moreover, we demonstrate that H-LTRF can meet all the properties in Section 3. The detailed proofs are given in Appendix C of the supplemental material, available online. Finally, Table 2 summarizes the properties that are satisfied by MLRF, LTRF, and H-LTRF, respectively. We can see that MLRF is *not* suitable for cloud computing system due to its lack of support for three important desired properties. In contrast, LTRF and H-LTRF, are *suitable* for cloud computing system.

6 LTYARN: A LONG-TERM YARN FAIR SCHEDULER

We have implemented the fair resource allocations in YARN by proposing a new scheduler called LTYARN.

6.1 Long-Term Max-Min Fairness Model

This section proposes long-term max-min fairness model for LTYARN. YARN is a hierarchical tree structure of multi-level fairness. The following part considers the bottom-level (i.e., application-level). The mechanism can be easily extended to upper queue-level.

Let $\Lambda = \{\Lambda_1, \Lambda_2, \Lambda_3, \ldots\}$ denote the set of submitted applications, and $\widetilde{\Lambda}$ be the set of its active applications (the '*active*' means there are pending or running tasks available). Let $n_i^p(t)$ denote the number of pending (i.e., runnable) tasks for the application Λ_i at time t. Let ω_i be the shared weight for the ith application. Based on the weighted max-min fairness strategy and Formula (1), the application Λ_i to be chosen at time t for fair resource allocation should satisfy the following condition:

$$\frac{f_i(t)}{\omega_i} = \min_{\Lambda_k \in \widetilde{\Lambda}} \left\{ \frac{f_k(t)}{\omega_k} \mid n_i^p(t) > 0 \right\}. \tag{6}$$

6.2 Design and Implementation of LTYARN

Fig. 3 overviews the design and implementation of LTYARN. It consists of three key components: *Quantum Updater (QU)*, *Resource Controller (RC)*, and *Resource Allocator (RA)*. QU is
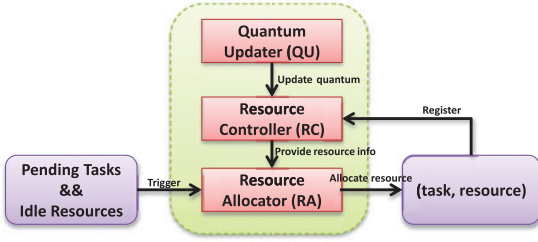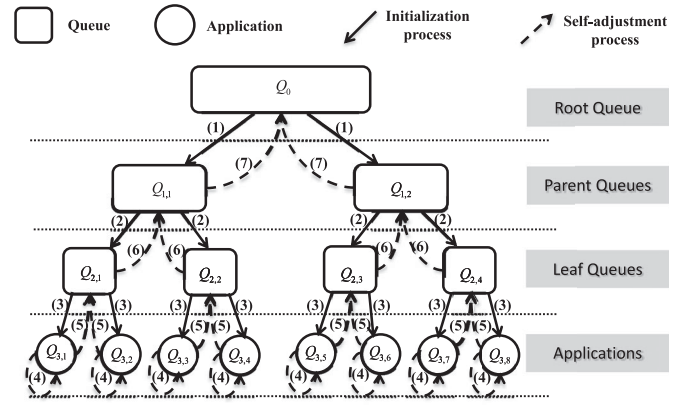
Fig. 3. Overview of LTYARN.

Fig. 4. The adaptive task quantum policy for YARN. The *up-to-bottom* data flow is a task time quantum initialization process for new applications. The *bottom-to-up* data flow is a quantum self-adjustment process for existing applications/queues.

responsible for updating the time quantum for each queue dynamically, based on running and completed tasks, to solve the unknown execution time issue. RC manages the allocated resources for each application/queue and computes the accumulated resources periodically. RA performs the resource allocation based on the accumulated resources of each application/queue.

6.2.1 Quantum Updater

Our long-term max-min fairness policy of LTYARN is based on the *accumulated* resources. When estimating the *accumulated* amount of resource for a task, we need to know the demand of its requested resources and the execution time it takes.

The resource demand can be obtained from the user request when the task is submitted to YARN. However, for *online* applications, the task execution time is generally unknown in advance. To address this problem, we propose a *Time Quantum-based Approach*, which is an approximation method. It gives a concept of *assumed execution time* denoted as Q, to represent the prior unknown *real* execution time. It is initialized with a time quantum threshold and can be adjusted dynamically to make it close to the real execution time based on the completed tasks. Specifically, our approach works as follows. We first initialize the *assumed execution time* to be zero for any *pending* task. When a task starts running, we give a time quantum threshold for its *assumed execution time*. For each running task, when its running time exceeds the *assumed execution time*, the *assumed execution time* is updated to the running time. In contrast, for any finished task, its *assumed execution time* is updated to its running time, no matter it is larger or smaller than the time threshold.

However, in practice, different applications often have different task execution time. Thus, a single static value of Q cannot meet the suitability requirements of multiple jobs (applications) at the same time. In addition, due to the possibility of varied types of applications in different queues for YARN in practice, ensuring that each queue owns a suitable Q for its own applications is necessary so that they do not interfere with each other.

With the above concerns, we propose an adaptive task quantum policy, implemented in Quantum Updater. It is a multi-level self-tunning approach based on the customized structure of YARN's resource organization, as shown in Fig. 4. The up-to-bottom data flow is a quantum value assignment process. It works when a new element (e.g., queue or application) is added. In contrast, the bottom-to-up data flow is a self-tunning procedure, which refreshes periodically by a fixed time interval (e.g., 1 second).

Initially, the system administrator provides a threshold value for root-level quantum Q_0. When a new application

is submitted to the system, it will perform up-to-bottom initialization process. First, it will check whether its parent queue is new one or not (Arrow (1) in Fig. 4). If yes, it will assign its parent-queue quantum with root-queue quantum, e.g., $Q_{1,1} \leftarrow Q_0$. Next, it checks its sub-queues (e.g., leaf-queue) (Arrow (2) in Fig. 4). If it is a new one, it will assign its sub-queue quantum with its parent-queue quantum, e.g., $Q_{2,1} \leftarrow Q_{1,1}$. Lastly, it initializes its application quantum with its leaf-queue quantum, e.g., $Q_{3,1} \leftarrow Q_{2,1}$ (Arrow (3) in Fig. 4).

In our implementation, Quantum Updater checks the system periodically for new finished tasks. When there is a map/reduce task finished, the bottom-to-up self-adjustment process begins to work automatically. First, it will update its application quantum with its average task completion time (Arrow (4) in Fig. 4). Next, it updates its leaf-queue quantum with its average application quantum (Arrow (5) in Fig. 4). Similarly, it updates its parent-queue quantum using the average value of its leaf-queue quantum (Arrow (6) in Fig. 4). Finally, the root-queue quantum is updated with the average value of parent-queue quantum (Arrow (7) in Fig. 4).

6.2.2 Resource Controller

Resource Controller is the main component of LTYARN. Its principle responsibility is to manage and update the accumulated resources for each queue. It tracks the allocated resource (e.g., container in YARN) and the execution time for each task. Based on this information, it performs the resource updating work periodically (e.g., 1 second).

Time Window-Based Support. To make it flexible, instead of keeping the long-term resource fairness all the time since the system starts, our LTYARN supports the long-term fairness within a period of time (e.g., 1 hour). That is, we divide the whole time into a set of time windows and ensure the fairness within the window (Intra-window allocation). When a new window starts, the previous historical allocation information is dropped and it performs the long-term fair resource allocation from the beginning(Inter-window allocation). For simplicity, we use *round* to denote *time window* below.

Resource Tracking and Estimation. It works based on the long-term max-min fairness model in Section 6.1. Each time,

TABLE 3
Job Types and Sizes for Each Bin in Our Synthetic
Facebook Workloads

Bin	Job Type	# Maps	# Reduces	# Jobs
1	rankings selection	1	NA	38
2	grep search	2	NA	18
3	uservisits aggregation	10	2	14
4	rankings selection	50	NA	10
5	uservisits aggregation	100	10	6
6	rankings selection	200	NA	6
7	grep search	400	NA	4
8	rankings-uservisits join	400	30	2
9	grep search	800	60	2

RC first estimates the assumed execution time for each running/completed task with the updated quantum value from QU. Next, it computes and updates the accumulated resource for each application/queue.

6.2.3   Resource Allocator

Resource Allocator locates at each queue of different levels, as shown in Fig. 4. It is triggered whenever there are pending tasks and idle resources. RA can now support FIFO, memoryless max-min fairness and long-term max-min fairness for each queue. Users can choose either of them accordingly. For long-term max-min fairness, it performs fair resource allocation for each application/queue with the provided resource information from RC, based on Formula (6). We provide two important configuration arguments for each queue, e.g., time quantum Q and round length L in the default configuration file, to meet different requirements for different queues. Moreover, we also support minimum (maximum) resource share for queues under long-term max-min fairness.

7   EVALUATION

We ran our experiments in a local cloud environment, which is established in a cluster consisting of 10 compute nodes, each with two Intel X5675 CPUs (six CPU cores per CPU with 3.07 GHz), 24 GB DDR3 memory. We emulate the $t2.medium$ instances of Amazon EC2 by configuring ⟨2 cores, 4 GB⟩ per VM and thereby create 6 VMs per node. The Hadoop-2.2.0 is chosen in our experiment. We configure 1 instance as master, and the remaining 59 instances as slaves.

Our evaluation methodology is as follows. We first construct a single-level hierarchy for LTYARN with the four macro-workloads below. We compare LTYARN with the default Hadoop Fair Scheduler (HFS). Second, we construct a two-level hierarchy for LTYARN by grouping four macro-workloads into groups to assess our design on hierarchical resource allocations.

7.1   Macro-Benchmarks

To evaluate our long-term fair scheduler LTYARN for YARN, we ran a macro-benchmark consisting of four different workloads:

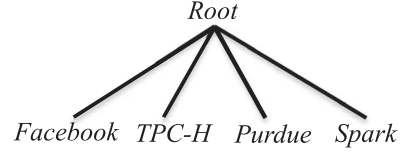- A MapReduce instance with a mix of small and large jobs based on the workload at the Facebook.

Fig. 5. Single-level hierarchical for LTYARN. Four different leaf queues are configured, i.e., *Facebook*, *Purdue*, *Spark*, *TPC-H*, corresponding to each workload, respectively.

- A MapReduce instance running a set of large-size batch jobs generated with Purdue MapReduce Benchmarks Suite [33].
- Hive [25] running a series of TPC-H queries.
- Spark [13] running a series of machine learning applications.

7.2   Macro-Workloads

Synthetic facebook workload. We synthesize our Facebook workload based on the distribution of jobs sizes and inter-arrival time at Facebook in Oct. 2009 provided by Zaharia et. al. [20]. The workload consists of 100 jobs. We categorize them into nine bins of job types and sizes, as listed in Table 3. It is a mix of large number of small-sized jobs (1 ∼ 15 tasks) and small number of large-sized jobs (e.g., 800 tasks[3]). The job submission time is derived from one of SWIM's Facebook workload traces (e.g., FB-2009_samples_24_times_1hr_1.tsv) [34]. The jobs are from Hive benchmark [35], containing four types of applications, i.e., rankings selection, grep search (selection), uservisits aggregation and rankings-uservisits join.

Purdue workload. We select five benchmarks (e.g., Word-Count, TeraSort, Grep, InvertedIndex, HistogramMovices) randomly from Purdue MapReduce Benchmarks Suite [33]. We use 40 G wikipedia data [36] for WordCount, Inverte-dIndex and Grep, 40G generated data for TeraSort and HistogramMovices with their provided tools. To emulate a series of regular job submissions in a data warehouse, we submit these five jobs sequentially at a fixed interval of 3 mins to the system.

TPC-H. To emulate continuous analytic query, such as analysis of users' behavior logs, we ran TPC-H benchmark queries on hive [14]. 40 GB data are generated with provided data tools. Four representative queries Q1, Q9, Q12, Q17 are chosen, each of which we create five instances. We write a script to launch each one after the previous one finished in a round robin fashion.

Spark. Latest version of Spark has supported its job to run on the YARN system. We consider two CPU-intensive machine learning algorithms, namely, kmeans and alternating least squares (ALS) with provided example benchmarks. We ran 10 instances of each algorithm, which are launched by a script that waits 2 minutes after each job completed to submit the next.

We configure a single-level hierarchy with the four workloads, as shown in Fig. 5. Each leaf queue corresponding to a workload. We use it for the following sections, i.e., Sections 7.3, 7.4, 7.5, and Appendix D of the supplemental material, available online. Besides, the hierarchy of more levels is given and used in Section 7.6.

---

3. We reduce the size of the largest jobs in [20] to have the workload fit our cluster size.

(a) Normalized current resource demand for each queue, with respect to its current share.

(b) Normalized currently allocated resources for each queue, with respect to its current share.

(c) Normalized accumulated resources for each queue, with respect to the system capacity.
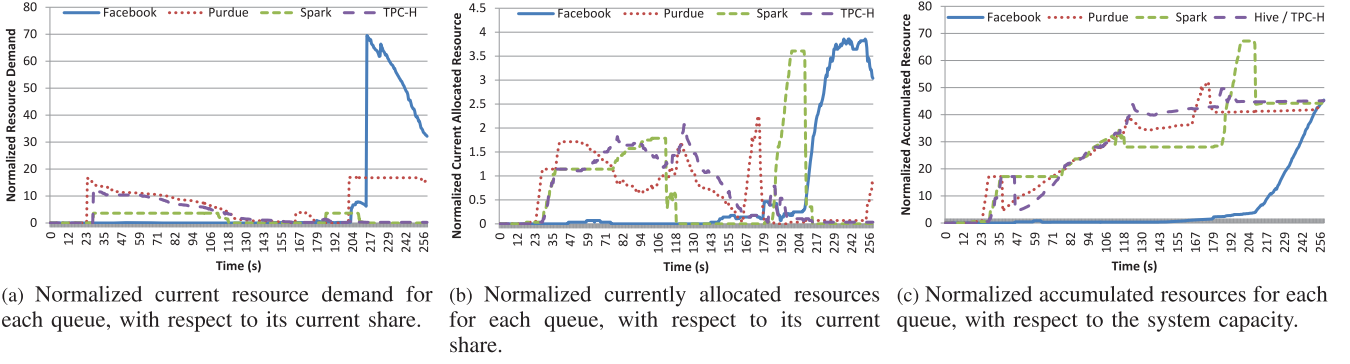
Fig. 6. Overview of detailed fairness resource allocation flow for LTYARN.

7.3 LTYARN Resource Allocation Flow

To understand the dynamic history-based resource allocation mechanism of LTYARN, we sample the resource demands, currently allocated resources and accumulated resources for four workloads over a short period of $0 \sim 260$ seconds, as illustrated in Fig. 6. Figs. 6a and 6b show the normalized results of the current resource demand and currently allocated resources for each workload with respect to its current share. Fig. 6c presents the normalized accumulated resources for four workloads with respect to the system capacity.

Fig. 6a shows that different workloads have different resource demands over time. At beginning, Purdue, Spark and TPC-H have an overloaded demand period (e.g., Purdue: 24-131, Spark: 28-118, TPC-H: 28-146). Fig. 6b shows the allocation details for each workload over time. We can see that, during the common overloaded period of 28-118, the curves for Purdue, Spark and TPC-H are fluctuated, indicating that LTYARN is dynamically adjusting the amount of resource allocation to each workload, instead of simply assigning each workload the same amount of resources like MLRF. Through

dynamic adjusting, the accumulated resources for the three workloads are balanced (i.e., the curve is close to each other) during the period 80-118, as shown in Fig. 6c. However, for Facebook workload, its overloaded period occurs from 204-260. During this period, the Purdue workload is also overloaded, as shown in Fig. 6a . To achieve the accumulated resource fairness, LTYARN allocated a large amount of resource to it (e.g., $3.85/4.0 = 96.25$ percent at point 222) shown in Fig. 6b, to make it catch up with others. We can see the accumulated resource results in Fig. 6c that, during 204-260, there is a significant increment for Facebook workload, whereas other workloads increase slightly.

7.4 Macrobenchmark Fairness Results

In Section 4.2, we have shown that a good sharing policy should be able to satisfy two key points: 1). can minimize the sharing loss (i.e., SLA guarantee) to make it close or equal to zero; 2). can maximize the sharing benefit as much as possible (i.e., Sharing incentive). This section makes a comparison between HFS and LTYARN on these two points.

We show the compared fairness results between HFS and LTYARN for four workloads over time in Fig. 7. All results are relative to the static partition case (without sharing) with fairness degree of one and sharing benefit/loss degrees of zero. Figs. 7a and 7c present the sharing benefit/loss degrees based on Formula (4) and (5), respectively, for HFS and LTYARN. Figs. 7b and 7d show the detailed fairness degree for each queue (workload) over time. We have the following observations:

First, the sharing policies of both HFS and LTYARN can bring sharing benefits for queues (workloads). For example, both Facebook and Purdue workloads, illustrated in Figs. 7b and 7d obtain benefits under the shared scenario. This is due to the sharing incentive property, i.e., each queue has an opportunity to consume more resources than her share at a time, better off running at most all of her shared partition in a non-shared partition system.

Second, it can also have the sharing loss problem easily for some queues if without a good sharing policy, i.e., whose total resources are worse than that in the non-shared case. In the shared computing system with SLA requirement, this is a very serious problem and thereby we should minimize the sharing loss as much as possible. The graph in Figs. 7a and 7c show that, LTYARN has a much better result than HFS. Specifically, Fig. 7a indicates that the sharing loss problem for HFS is constantly available until all the workloads complete (e.g., ≈ -0.5 on average), contributed

(a) Sharing benefit/loss degree with HFS based on Formula (4) and (5).

(b) Detailed fairness degree for four queues with HFS based on Formula (3).

(c) Sharing benefit/loss degree with LTYARN based on Formula (4) and (5).

(d) Detailed fairness degree for four queues with LTYARN based on Formula (3).
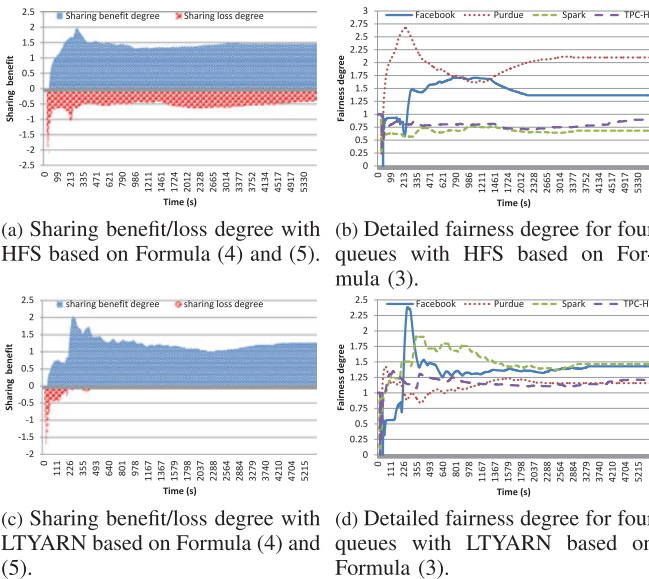
Fig. 7. Comparison of fairness results over time for each of workloads under HFS and LTYARN in YARN. All results are relative to the static partition scenario (without sharing) whose fairness degree is always one and sharing benefit/loss is zero. (a) and (c) show the overall benefit/loss relative to the non-sharing scenario. (b) and (d) present the detailed fairness degree for each queue: 1). A queue gets sharing benefit when its fairness degree is larger than one; 2). Otherwise, it arises sharing loss problem when a queue's fairness degree is below one.
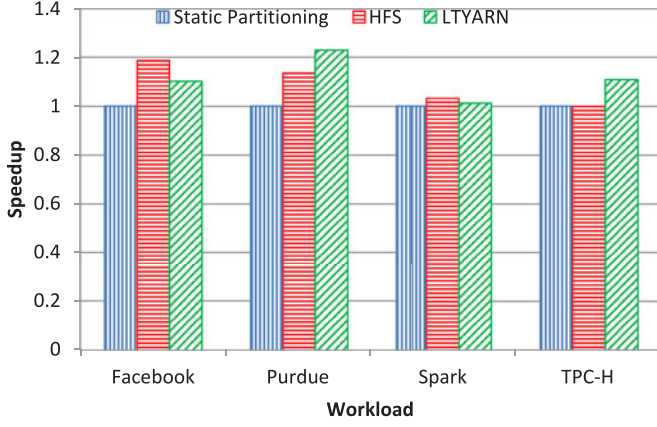
Fig. 8. The normalized performance results (e.g., speedup) for Static Partitioning, HFS and LTYARN, with respect to static partitioning.
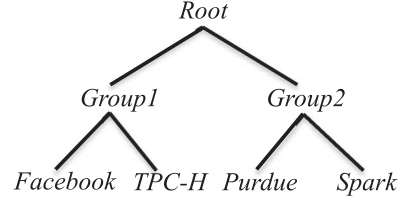

Fig. 9. The hierarchical structure of multi-level resource allocation for four workloads under two groups, i.e., *Group1* and *Group2*, where *Facebook* and *TPC-H* workloads are under *Group1*, *Purdue* and *Spark* workloads are under *Group2*.

primarily by Spark and TPC-H workloads given by Fig. 7b . In contrast, there is no more sharing loss problem after 650 seconds for LTYARN, i.e., all workloads get sharing benefits after that. The explanation is that, for MLRF, it does not consider historical resource allocation. Due to the varied demands for each workload over time, it easily occurs two extreme cases: 1). some workloads get much more resources over time (e.g., Facebook and Purdue workloads in Fig. 7b); 2). some workloads obtain much less resources that without sharing over time (e.g., Spark and TPC-H workloads in Fig. 7b). In contrast, LTYARN is a history-based fairness resource allocation policy. It can dynamically adjust the allocation of resources to each queue in terms of their historical consumption so that each queue can obtain much close amount of total resources over time. Its *lending agreement* can avoid two extreme cases existing in HFS.

Finally, regarding the sharing loss problem at the early stage (e.g., $0 \sim 650$ seconds) of LTYARN in Fig. 7c , it is mainly due to the unavoidable waiting allocation problem at starting stage: a first coming and running workload possess all resources and leads late arriving workloads need to wait for a while until some tasks complete and release resources (e.g., it is obvious in 0-226 seconds in Fig. 7c). The problem does also exist in HFS. But LTYARN can smooth this problem until it disappear over time via *lending agreement*, while HFS cannot.

7.5   Macrobenchmark Performance Results

Fig. 8 presents the performance results (i.e., speedup) for four workloads under Static Partitioning, HFS and LTYARN, respectively. All results are normalized with respect to Static Partitioning. We see that, 1). the shared cases (i.e., HFS and LTYARN) can possibly achieve better performance than the non-shared case (i.e., Static Partitioning), or at least as fast in the shared system as they do in their static partitioning system. For example, for Facebook and Purdue workloads, both HFS and LTYARN have much better performance results (e.g., $14 \sim 19$ percent improvement for HFS, and $10 \sim 23$ percent for LTYARN) than exclusively using a static partitioning system. The finding is consistent with previous works such as Mesos [10]. The performance gain is mainly due to the resource preemption of unneeded resources from other queues in a shared system. The statement can be validated by reviewing our fairness results in

Figs. 7b and 7d in Section 7.4. We can note that, the fairness degrees for both Facebook and Purdue workloads are above one (i.e., get sharing benefit) during the most of time, except at the beginning stage. For Spark and TPC-H workloads, HFS and LTYARN can be at least as fast as static partitioning. 2). There is no conclusive result regarding which one is absolutely better than the other between HFS and LTYARN. For example, HFS is better than LTYARN for Facebook by about 7 percent and Spark by about 2 percent. However, LTYARN outperforms HFS for Purdue workload by about 8 percent and TPC-H by about 10 percent.

7.6   Hierarchical Resource Allocation Evaluation

In this section, we evaluate the hierarchical resource allocation primarily from two points. First, we show that the naive approach can cause the starvation problem in resource allocation. Second, we demonstrate that the starvation problem can be addressed with proposed (starvation-aware) H-LTRF in Section 5.2.2.

We start by establishing a multi-level queue structure with the aforementioned four workloads in YARN, based on Fig. 1b in Section 5.2. The resulting hierarchical structure is illustrated in Fig. 9. We create two queues, namely, *Group1* and *Group2* with equal share of the whole system (i.e., *Root* node). For *Group1*, it has two children, i.e., *Facebook* and *TPC-H*. In contrast, there are two sub-queues, namely, *Purdue* and *Spark* under *Group2*. We assume that all leaf nodes under the same parent are with equal share. Moreover, to show the starvation phenomenon (e.g., for TPC-H workload) in hierarchical resource allocation, we submit the Facebook workload first, and after 15 minutes we then submit other three workloads (i.e., Purdue, Spark, TPC-H) simultaneously.

Figs. 10a and 10b present the memory resource allocation results for queues at different levels under the naive approach. At the first time period of 900 seconds, only *Facebook* workload is running, which contributes to *Group1* of the accumulated resource. After 900th second, other three workloads are submitted. Based on the naive approach, it allocates all idle resources to *Group2* so that the accumulated resource of Group2 can catch up with that of *Group1* quickly. We can witness from Fig. 10a that there is no resource allocation for *Group1* from the 900th second to $1,580$th second (i.e., ≈ 11 minutes). Fig. 10b gives the detailed resource allocation for four workloads over time. We can see that, because of recursively collective resource allocation, *TPC-H* workload has been starved for at least 11 minutes without being allocated resources since its submission at the 900th second. This is due to the accumulated resource consumption of *Group1* contributed by its sibling *Facebook* workload at the starting period of 900 seconds.

(a) The resource allocation for two groups over time under the naive approach.

(b) The detailed resource allocation for four workloads over time under the naive approach.

(c) The comparison results of resource allocation for the TPC-H workload under H-LTRF of different timeout T_{wait}.
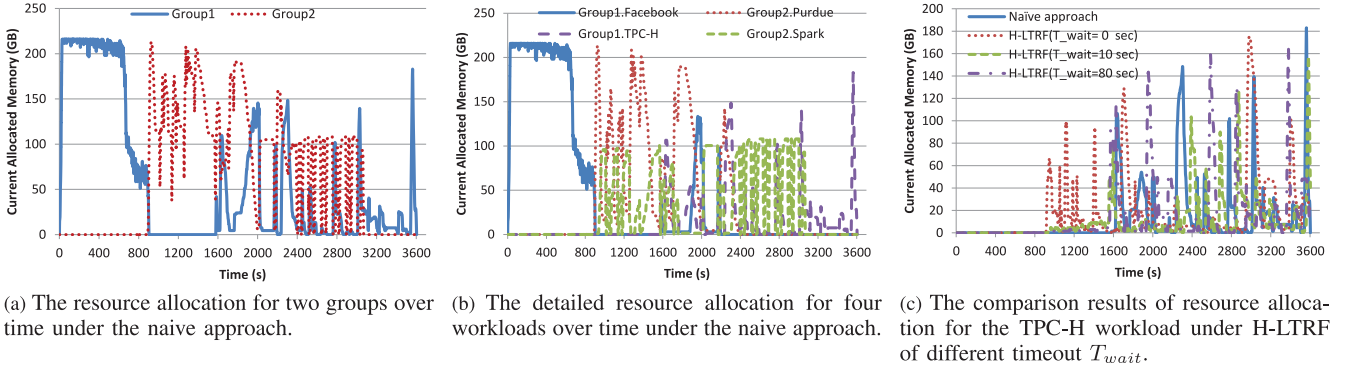
Fig. 10. The comparison of hierarchical resource allocation under naive approach and (starvation-aware) H-LTRF subject to Fig. 9. To show the starvation problem (e.g., for TPC-H workload), we perform the experiment by submitting Facebook workload first. After 15 minutes, we submit other three workloads (i.e., Purdue, Spark and TPC-H) simultaneously. Figs. 10a and 10b shows that TPC-H workload (i.e., the sibling of Facebook workload) has been starved for at least 680 seconds (i.e., ≈ 11 minutes) without being allocated resources since its submission at the 900th second.

Fig. 10c presents the comparison results of resource allocation for *TPC-H* under the naive approach and (starvation-aware) H-LTRF of different timeout T_{wait}. We have the following observations:

First, our H-LTRF can alleviate the starvation problem in recursively collective resource allocation significantly. In comparison to the naive approach under which the TPC-H workload is starved for at least 11 minutes without being allocated resources since its submission, our proposed H-LTRF can alleviate this problem by ensuring that TPC-H workload is not starved for T_{wait} period of time since its last time resource allocation. For example, when $T_{wait} = 10$ seconds, we mean that TPC-H workload will not be starved for 10 seconds since its last time resource allocation.

Second, H-LTRF is able to allow users to balance the tradeoff between the resource lending affinity and starvation reduction in hierarchical resource allocation by setting the argument T_{wait} flexibly according to their needs. We can observe in Fig. 10c that the TPC-H workload is actively allocated resources since its submission at 900th second when $T_{wait} = 0$ (i.e., no starvation problem here). When $T_{wait} = 10$ seconds, there are $2 \sim 10$ actively running tasks of TPC-H at during its starvation period (from 900th to 1,580th seconds). In contrast, there are $1 \sim 4$ actively running tasks at that period for large value of $T_{wait} = 80$.

By far, one remaining issue for H-LTRF is about T_{wait} configuration. We see in Fig. 10c that its value can have a significant impact on the average number of actively running tasks for those starved workloads during their starvation periods. To understand it theoretically, for simplicity, consider a workload whose tasks are of the same duration T_{task}. During the starvation period, H-LTRF will schedule its tasks every T_{wait} seconds, then it holds for the number of active running tasks $N_{task} = T_{task}/T_{wait}$, assuming that there are enough resources. In practice, users could configure T_{wait} based on the average task duration T_{avg}, i.e.,

$$T_{wait} = \frac{T_{avg}}{N_{task}}. \qquad (7)$$

Fig. 11a presents the task execution time for 1,000 tasks of TPC-H workload, whose average execution time is 23.6 seconds. Fig. 11b shows the corresponding number of actively running tasks for different values of T_{wait} over time. Based on Formula (7), we see that there are $\frac{23.6}{2} \approx 12$ actively running tasks on average when $T_{wait} = 2$. Similarly, when $T_{wait} = 4$ and 10, there are on average $\frac{23.6}{4} \approx 6$ and $\frac{23.6}{10} \approx 3$ actively running tasks, respectively.

8 RELATED WORK

We review the literature works that are close to us from the following three perspectives:

● *Data-intensive computing in the cloud*. The study on cloud-based data-intensive computing has been a hot research area in recent years. There have been many studies on the performance optimization for diverse workloads and data (e.g., [17], [18], [19], [31], [49]) under various data-intensive computing frameworks (e.g., MapReduce [2], Spark [13], HIVE [25], Storm [12]). Considering that the cloud resources are charged on the basis of pay-as-you-use model, a number of studies become available on the monetary cost minimization (e.g., [4], [9]). Moreover, there are some other interesting works focusing on data privacy [39] and reliability [11]. In contrast, this paper considers the fairness resource allocation issue in the cloud, where YARN is specifically considered. We show that there are three problems for existing fair scheduler of YARN, making it *not* suitable for cloud computing. To address it, we have proposed a new scheduler called LTYARN.

● *Fairness definitions, policies and algorithms*. Fairness has been studied extensively with various ways [26], [27], [29], [30], [32]. The fairness they defined are mainly based on the "performance" metrics such as start time, response time, wait time and slowdown. For example, Zhao and

(a) Task execution time for 1000 tasks of TPC-H workload, whose average execution time is 23.6 seconds.

(b) The number of running tasks for TPC-H workload under different timeout T_{wait} over time of unit time (i.e., T_{wait}).
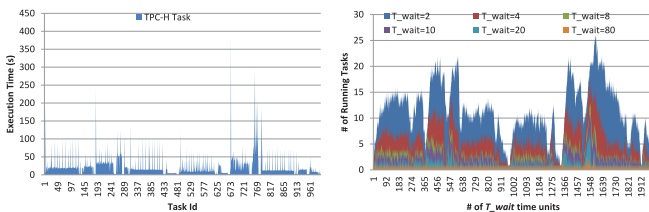
Fig. 11. The impact of T_{wait} on the number of actively running tasks for TPC-H workload.

Sakellariou [30] and Arabnejad and Barbosa [32] consider the fairness for multiple workflows. They define fairness on the basis of *slowdown* that each workflow would experience, where the *slowdown* refers to the difference in the expected execution time for the same workflow between when scheduled together with other workflows and when scheduled alone. However, we show that they are no longer suitable for cloud computing systems due to the different concerns and meanings of fairness preferred in cloud computing systems as follows:

First, the cloud computing system is a service-oriented platform with resource guarantee. That is, from service providers' perspective (e.g., Amazon, supercomputer operator), they only need to guarantee the amount of resources allocated to each user over a period of time. In comparison, the performance metrics for user's applications are not the main concerns for providers. Our proposed LTRF and its extension H-LTRF are based on this point. They attempt to make sure that the total amount of resources that each user obtained in the shared cloud system is larger than or at least the same as that in an non-shared partitioning system, according to the users' resource contributions.

Second, the traditional fair policies and algorithms (e.g., round-robin, proportional resource sharing [37], and weighted fair queueing [38]) on resource allocation are *memoryless*, i.e., instant fairness of one dimension. Our LTRF is designed to be a two-dimension fair policy with the historical information considered.

• *Max-min fairness and supporting systems*. In the literature, there are many work that apply max-min fairness to a variety of single-typed resources, including CPU [21], [37], memory [16], [37], network bandwidth [42], [44], and storage [45]. For cloud computing, the max-min fairness has been widely used by many data-intensive systems, such as Hadoop [15], YARN [3], Mesos [10], Choosy [8], and Pisces [45].

Besides, there are lots of work extending the max-min fairness for multi-resource allocation. Dominant Resource Fairness [23] is the first work in the literature for multi-resource allocation. It achieves the fairness by equalizing the dominant shares (i.e., the highest share of any typed resource allocated) across all users. DRF has been implemented in many current datacenter frameworks such as YARN [3] and Mesos [10]. After that, there have been lots of extension and generalization for DRF, including [28], [41], [50].

Despite the various efforts, all of these fair policies and their implementation on existing systems above are indeed *memoryless* (i.e., allocating resources fairly at instant time), belonging to MLRF. We show that there are three problems for cloud computing system regarding MLRF, i.e., cost-inefficient workload submission problem, strategy-proofness problem and resource-as-you-contributed unfairness problem. This paper goes beyond our previous work [40], [51] in the following aspects: (I). We provide more details of LTRF for single-resource allocation; (II). We extend LTRF for hierarchical resource allocation. (III). We perform more extensive experiments in our evaluation.

9 CONCLUSION

This paper studies the resource allocation fairness for YARN in cloud environment. We find that, the existing max-min fair policy used in YARN, is *not* suitable in cloud

computing system. We show that this is because of its *memoryless* resource allocation manner that can cause three serious problems, i.e., cost-inefficient workload submission problem, strategy-proofness problem and resource-as-you-contributed unfairness problem. To address these problems for YARN, we propose LTRF and H-LTRF for single-level and hierarchical resource allocation, respectively. We demonstrate that they are *suitable* for cloud computing system. Finally, we developed *LTYARN*, a long-term YARN fair scheduler for YARN and our experimental results validate the effectiveness of our solutions. We have made LTYARN open source at *http://sourceforge.net/projects/ltyarn/*.

REFERENCES

[1] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: Mapreduce-style processing of fast data," *Proc. VLDB Endow.*, 2012, vol. 5, pp. 1814–1825.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," presented at the 6th Symp. Operating System Design and Implementation, San Francisco, CA, USA, 2004.
[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, et al., "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[4] M. Ming and H. Marty, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–12.
[5] H. Y. Wang, Q. F. Jing, R. S. Chen, B. S. He, Z. P. Qian, and L. D. Zhou, "Distributed systems meet economics: Pricing in the cloud," in *Proc. HotCloud*, 2010, pp. 6–12.
[6] M. Isard, M. Budiu, Y. Yu, A. Birell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.
[7] A. Fight. *Introduction to Project Finance*. London, U.K.: Butterworth-Heinemann, 2005.
[8] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 365–378.
[9] C. Zhou and B. S. He, "Transformation-based monetary cost optimizations for workflows in the cloud," *IEEE Trans. Cloud Comput.*, vol. 2, no. 1, pp. 85–98, Jan./Mar. 2014.
[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, Mar. 2011, pp. 295–308.
[11] Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, B. Tudor, "Fast failure recovery in distributed graph processing systems," *Proc. VLDB Endow.*, vol. 8, pp. 437–448, 2015.
[12] (2016). Apache. *Storm* [Online]. Available: http://storm-project.net/.
[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–16.
[14] Apache. *TPC-H Benchmark on Hive* [Online]. Available: https://issues.apache.org/jira/browse/HIVE-600.
[15] (2016). Apache. *Hadoop* [Online]. Available: http://hadoop.apache.org.
[16] A. K. Agrawala and R. M. Bryant, "Models of memory scheduling," in *Proc. 5th ACM Symp. Operating Syst. Principles*, 1975, pp. 217–222.
[17] L. Qin, J. X. Yu, L. J Chang, H. Cheng, C. Q. Zhang, and X. M. Lin, "Scalable big graph procing in mapreduce," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2014, pp. 827–838.

[18] A. Turk, R. O. Selvitopi, H. Ferhatosmanoglu, and C. Aykanat, "Temporal workload-aware replicated partitioning for social networks," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2832–2845, Nov. 2014.

[19] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data procing," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 1–13.

[20] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Schenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSyst.*, 2010, pp. 265–278.

[21] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[22] (2016). Apache. *Hadoop MapReduce Next Generation-Fair Scheduler* [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Schenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 24–37.

[24] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," *in Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–15.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, and H. Liu, "Hive—A petabyte scale data warehouse using hadoop," in *Proc. IEEE Conf. Data Eng.*, 2010, pp. 996–1005.

[26] G. Sabin, G. Kochhar, and P. Sadayappan, "Job fairness in non-preemptive job scheduling," in *Proc. Int. Conf. Parallel Process.*, 2004, pp. 186–194.

[27] R. Jain, D. M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer system," Digital Equipment Corporation, 77 Reed Road Hudson, Ma 01749, Tech. Rep. EC-TR-301, 1984.

[28] H. K. Liu and B. S. He, "Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in IAAS clouds," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 970–981.

[29] J. Ngubiri and M. V. Vliet. "A metric of fairness for parallel job schedulers," *J. Concurrency Comput: Practice Experience*, vol. 21, pp. 1525–1546, 2009.

[30] H. N. Zhao and R. Sakellariou, "Scheduling multiple DAGs onto heterogeneous systems," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006, pp. 159–172.

[31] M. A. Bhuiyan and M. A. Hasan, "An iterative mapreduce based frequent subgraph mining algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 3, pp. 608–620, Aug. 2014.

[32] H. Arabnejad and J. Barbosa. "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl.*, 2012, pp. 633–639.

[33] F. Ahmad, S. Y. Lee, M. Thottethodi, and T. N. Vijaykumar, "PUMA: Purdue mapreduce benchmarks suite," Purdue University, West Lafayette, IN 47907-1285, ECE Tech. Rep., 2012.

[34] (2012). GitHub. *Facebook workload traces* [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[35] (2009). Apache. *Hive performance benchmarks* [Online]. Available: https://issues.apache.org/jira/browse/HIVE-396.

[36] (2016). PUMA Datasets [Online]. Available: https://engineering.purdue.edu/~puma/datasets.htm.

[37] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proc. 1st USENIX Conf. Operating Syst. Des. Implementation*, 1994, pp.1–11.

[38] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," in *Symp. Proc. Commun. Archit. Protocols*, 1989, pp. 1–12.

[39] N. Cao, C. Wang, M. Li, and K. Ren, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, Jan. 2014.

[40] S. J Tang, B. S. Lee, B. S. He, and H. K. Liu, "Long-Term resource fairness: Towards economic fairness on pay-as-you-use computing systems. in *Proc. 28th ACM Int. Conf. Supercomput.*, 2014, pp. 251–260.

[41] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet procing," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 1–12.

[42] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 187–198.

[43] F. Fassetti, G. Greco, and G. Terracina, "Mining loosely structured motifs from biological data," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 11, pp. 1472–1489, Nov. 2008.

[44] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. OShea, "Chatty tenants and the cloud network sharing problem," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, pp. 171–184.

[45] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 349–362.

[46] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, and Q. Xu, "Es 2: A cloud data storage system for supporting both OLTP and olap," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 291–302.

[47] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. "The cost of a cloud: Research problems in data center networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 68–73, 2009.

[48] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QOS-aware cluster management," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 127–144.

[49] R. S. Chen, M. Yang, X. T. Weng, B. Choi, B. S. He, and X. M. Li, "Improving large graph processing on partitioned graphs in the cloud," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–13.

[50] H. K. Liu and B. S. He, "F2C: Enabling fair and fine-grained resource sharing in multi-tenant IaaS clouds," *IEEE Trans. Parallel Distrib. Syst.*, 2015, pp. 1–14.

[51] S. J. Tang, B. S. Lee, and B. S. He, "Towards economic fairness for big data processing in pay-as-you-go cloud computing," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Technol. Sci.*, 2014, pp. 638–643.
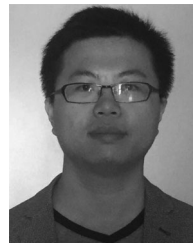
Shanjiang Tang received the BS and MS degrees from Tianjin University (TJU), Tianjin, China, in July 2008 and January 2011, respectively, and the PhD degree from the School of Computer Engineering, Nanyang Technological University, Singapore, in 2015. He is currently an assistant professor in the School of Computer Science and Technology, Tianjin University, China. His research interests include parallel computing, cloud computing, big data analysis, and computational biology.

Bu-Sung Lee received the BSc (Hons.) and PhD degrees from the Electrical and Electronics Department, Loughborough University of Technology, United Kingdom, in 1982 and 1987, respectively. He is currently an associate professor with the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include computer networks protocols, distributed computing, network management, and Grid/Cloud computing.

Bingsheng He received the Bachelor's degree in computer science from Shanghai Jiao Tong University, Shanghai, China (1999-2003), and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong (2003-2008). He is an associate professor at the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems.