Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems

Shanjiang Tang, Bu-Sung Lee, Bingsheng He, Haikun Liu
School of Computer Engineering, Nanyang Technological University
{stang5, ebslee, bshe}@ntu.edu.sg, haikunliu@gmail.com

ABSTRACT

Fair resource allocation is a key building block of any shared computing system. However, *MemoryLess Resource Fairness (MLRF)*, widely used in many existing frameworks such as YARN, Mesos and Dryad, is *not* suitable for pay-as-you-use computing. To address this problem, this paper proposes *Long-Term Resource Fairness (LTRF)*, a novel fair resource allocation mechanism. We show that LTRF satisfies several highly desirable properties. First, LTRF incentivizes clients to share resources via group-buying by ensuring that no client is better off in a computing system that she buys and uses individually. Second, LTRF incentivizes clients to submit *non-trivial* workloads and be willing to yield unneeded resources to others. Third, LTRF has a resource-as-you-pay fairness property, which ensures the amount of resources that each client should get according to her monetary cost, despite that her resource demand varies over time. Finally, LTRF is strategy-proof, since it can make sure that a client cannot get more resources by lying about her demand. We have implemented LTRF in YARN by developing *LT-YARN*, a long-term YARN fair scheduler, and shown that it leads to a better resource fairness than other state-of-the-art fair schedulers.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling; D.2.8 [**Metrics**]: Process metrics, performance measures; k.6.2 [**Installation Management**]: Pricing and resource allocation

Keywords

Cloud Computing, Long-Term Resource Fairness, MapReduce, YARN

1. INTRODUCTION

Current supercomputers and data centers (e.g., Amazon EC2) typically consist of thousands of servers connected via a high-speed network. At any time, there are tens of thousands of clients concurrently running their high-performance computing applications (e.g., MapReduce [8], MPI, Spark [32]) on the shared computing system (i.e., pay-as-you-use computing system). Clients pay the money on the basis of their resource usage. To meet different clients' needs, providers generally offer several options of price plans (e.g., on-demand and reservation). When a client has a short-term computation requirement (e.g., several hours), she can choose on-demand price plan that charges compute resources by each time unit (e.g., hour) with fixed price. In contrast, if she has a long-term computation request (e.g., 1 year), choosing reserved price plan can enable her to have a significant discount from the on-demand hourly charge and thereby save the money cost.

Instead of purchasing and utilizing resources individually, recently, there are some researchers and companies (e.g., Tuangru, SalesForce) strongly recommending *group-buying* and *resource sharing*, since *group-buying* can offer resources at significantly reduced prices on the condition that a minimum number of buyers would make the purchase [13] and *resource sharing* can improve the resource utilization. Consider buying the reserved resources for example. With reservation plan, clients need to pay a one-time fee for a long time (e.g., 1 or 3 years). To achieve the full cost savings, customers must commit to have a high utilization. In practice, it is most likely that the resource demand of a customer varies over time, indicating that it's difficult to ensure the resources can be fully utilized all the time.

With group-buying and resource sharing, the above problems can be nicely addressed. First, group buying can get increased discount of reserved resources from sellers, cheaper than buying individually. Second, different clients often have different resource demand at different time. The resource utilization problem can be thereby resolved with resource sharing between clients in a shared system.

Given group-buying resources, the fair resource allocation among clients is a key issue. One of the most popular fair allocation policy is *(weighted) max-min fairness* [11], which maximizes the minimum resource allocation obtained by a user in a shared computing system. It has been widely used in many popular high performance computing frameworks such as Hadoop [4], YARN [2], Mesos [15], Dryad [16] and Choosy [10]. Unfortunately, we observe that the fair polices implemented in these systems are all *memoryless*, i.e., allocating resources fairly at instant time without considering history information. We refer those schedulers as *MemoryLess Resource Fairness (MLRF)*. MLRF is *not* suitable for such pay-as-you-use computing system due to the following reasons.

Trivial Workload Problem. In a pay-as-you-use computing system, we should have a policy to incentivize group members to submit *non-trivial* workloads that they really need (See *Non-Trivial-Workload Incentive* property in Section 3). 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 real world. It can cause trivial workload problem with *MLRF*. Consider two users A and B sharing a system. Let D_A and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'14, June 10–13 2014, Munich, Germany.
Copyright 2014 ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597672.

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. In that case, it is possible that A is selfish and will try to possess all of her share by running some trivial tasks (e.g., running some duplicated tasks of the experimental workloads for double checking) so that her extra unused share will not be preempted by B, causing the inefficiency problem of running non-trivial workloads and also breaking the sharing incentive property (See the definition in Section 3).

Strategy-Proofness Problem. It is important for a shared system to have a policy to ensure that no group member can get any benefits by lying (See *Strategy-proofness* in Section 3). We argue that *MLRF* cannot satisfy this property. Consider a system consisting of three users A, B, and C. Assume A and C are honest whereas B is not. It could happen at a time that both the *true* demands of A and B are less than their own shares while C's *true* demand exceeds its share. In that case, A yields her unused resources to others honestly. But B will provide *false* information about her demand (e.g., far larger than her share) and compete with C for unused resources from A. Lying benefits B, hence violating strategy-proofness. Moreover, it will break the sharing incentive property if all other users also lie.

Resource-as-you-pay Fairness Problem. For group-buying resources, we should ensure that the total resource received by each member is proportional to her monetary cost (See *Resources-as-you-pay Fairness* in Section 3). Due to the varied resource demands (e.g., workflows) for a user at different time, *MLRF* cannot achieve this property. 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 B (i.e., lend to B) according to the work conserving property of *MLRF*. Next at time t_1, assume that A's demand D_A becomes larger than its share. With *MLRF*, user 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 *memoryless*. If this scenario often occurs, it will be unfair for A to get the amount of resources that she should have obtained from a long-term view. (See a motivation example in Section 4).

In this paper, we propose *Long-Term Resource Fairness (LTRF)* and show that it can solve the aforementioned problems. LTRF satisfies five good properties including sharing incentive, non-trivial-workload incentive, resource-as-you-pay fairness, strategy-proofness and Pareto Efficiency. *LTRF* provides incentives for users to submit non-trivial workloads and share resources via group-buying by ensuring that no customer is better off in a computing system that she purchases individually. Moreover, *LTRF* can guarantee the amount of resources a user should receive in terms of the monetary cost that she pays, in the case that her resource demand varies over time. In addition, *LTRF* is strategy-proof, as it can make sure that a customer cannot get more resources by lying about her resource demand. Finally, *LTRF* can maximize the system utilization by ensuring that it is impossible for a client to get more resources without decreasing the resource of at least one client.

We have implemented *LTRF* in YARN [2] by developing a long-term fair scheduler *LTYARN*. The experiments show that, 1). LTRF can guarantee SLA via minimizing the sharing loss and bringing much sharing benefit for each client, whereas MLRF cannot; 2). the shared methods using LTRF can 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 [15].

[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.

This paper is organized as follows. Section 2 reviews the related work. Section 3 gives several payment-oriented resource allocation properties. Section 4 presents LTRF and gives a property analysis, followed by the design and implementation of *LTYARN* in Section 5. Section 6 evaluates the fairness and performance of LTYARN experimentally. Finally, we conclude and give future work in Section 7.

2. RELATED WORK

We review the existing studies that are closely related to this work from two aspects below:

Fairness Definitions, Policies and Algorithms. Fairness has been studied extensively in HPC and grid computing environment [23, 18, 22, 34, 6]. Sabin et al. [23] consider fair in terms of start time, if no later arriving job delays an earlier arriving job. Jain et al. [18] measured the fairness based on the standard deviation of the turnaround time. Ngubiri et al. [22] compare different fairness definitions on dispersion, start time and queueing time. Zhao et al. [34] and Arabnejad et al. [6] consider 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 between when scheduled together with other workflows and when scheduled alone.

The above fairness definitions are mainly based on the "performance" metrics. In this following, we argue that they are no longer suitable due to the different concerns and meanings of fairness preferred in pay-as-you-use computing systems.

1). The pay-as-you-use 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 client over a period of time. That is, the performance metrics for client's applications are not the main concerns for providers. Our proposed LTRF is based on this point in the shared pay-as-you-use computing system. It attempts to make sure that the total amount of resources that each client obtains is larger than or at least the same as that in an non-shared partitioning system, according to her payment.

2). The traditional fair policies and algorithms (e.g., round-robin, proportional resource sharing [29], and weighted fair queueing [9]) on resource allocation in HPC and grid computing are *memoryless*, i.e., instant fairness of a single dimension. In contrast, pay-as-you-use computing system has a monetary cost issue with resources paid by consumed time (e.g., one hour). Its fair policy should have two dimensions, i.e., the size of resources multiplies the execution time that a client consumed. Our LTRF is designed to be a two-dimension fair policy with the historical information considered.

Max-Min Fairness. Max-min fairness is a popular fair policy widely used in many existing systems such as Hadoop [4], YARN [2], Mesos [15], Choosy [10], Quincy [17]. Hadoop [4] partitions resources into map/reduce slots and allocates them fairly across pools and jobs. In contrast, YARN [2] divides resources into containers (i.e., a set of various resources like memory and CPU cores) and tries to guarantee fairness among queues. Mesos [15] enables multiple diverse computing frameworks such as Hadoop and Spark sharing a single system. Choosy [10] extends the max-min fairness by considering placement constraints. Quincy [17] is a fair scheduler for Dryad that achieves the fair scheduling of multiple jobs by formulating it as a min-cost flow problem. Moreover, DRF [11] and its extensions [7, 19, 31, 25] generalize max-min fairness from a single resource type to multiple resource types. However, all of these are indeed *memoryless*, belonging to MLRF. In this paper, we argue that there are three problems in pay-as-

you-use computing system regarding MLRF, i.e., trivial workload, strategy-proofness and resource-as-you-pay. In contrast, our proposed LTRF can address all those three problems.

3. PAYMENT-ORIENTED RESOURCE ALLOCATION PROPERTIES

This section presents a set of desirable properties that we believe any payment-oriented resource allocation policy in a shared pay-as-you-use system should meet. Base on these properties, we design our fair allocation policy in the following sections. We have found the following five important properties:

- *Sharing Incentive*: Each client should be better off sharing the resources via group-buying with others, than exclusively buying and using the resources individually. Consider a shared pay-as-you-use computing system with n clients over t period time. Then a client should not be able to get more than $t \cdot \frac{1}{n}$ resources in a system partition consisting of $\frac{1}{n}$ of all resources.
- *Non-Trivial-Workload Incentive*: A client should get benefits by submitting non-trivial workloads and yielding unused resources to others when not needed. Otherwise, she may be selfish and posses all unneeded resources under her share by running some dirty or trivial tasks in a shared computing environment.
- *Resource-as-you-pay Fairness*: The resource that a client gains should be proportional to her payment. This property is important as it is a resource guarantee to clients.
- *Strategy-Proofness*: Clients should not be able to get benefits by lying about their resource demands. This property is compatible with sharing incentive and resource-as-you-pay fairness, since no client can obtain more resources by lying.
- *Pareto Efficiency*: In a shared resource environment, it is impossible for a client to get more resources without decreasing the resource of at least one client. This property can ensure the system resource utilization to be maximized.

4. LONG-TERM RESOURCE FAIRNESS

In this section, we first give a motivation example to show that MemoryLess Resource Fairness (MLRF) is *not* suitable for pay-as-you-use computing system. Then we propose Long-Term Resource Fairness (LTRF), a payment-oriented allocation policy to address the limitations of MLRF and meet the desired properties described in Section 3. Lastly, we introduce our formal fairness definition.

Motivation Example. Consider a shared computing system consisting of 100 resources (e.g., 100GB RAM) and two users A and B with equal share of 50GB each. As illustrated in Table 1, assume that the new requested demands at time t_1, t_2, t_3, t_4 for client A are $20, 40, 80, 60$, and for client 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 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 pseudo-code for LTRF scheduling. It considers the fairness of total allocated resources consumed by each client, instead of currently allocated resources. The core idea is based on the '*loan(lending) agreement*' [20] with free interest. That is, a client 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-client example with LTRF in Table 1(b), client A first lends her unused resources of 30 and 10 to client 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.

Due to the *lending agreement* of LTRF, in practice, when A yields her unused resources at t_1 and t_2, B might not want to possess extra unused resources from A immediately. In that case, the total allocations for A and B will be $160(= 20+40+50+50)$ and $200(= 50+50+50+50)$ at time t_4, causing the inefficiency problem for the system utilization. To solve this problem, we propose a *discount*-based approach. The idea is that, anybody possessing extra unused resources from others will have a *discount* (e.g., 50%) on resource counting. It will incentivize B to preempt extra unused resources from A, since it is *cheaper* than its own share of resources. For A, it also does not get resource lost, as it can get the same *discount* on the resource counting for the preempted resources from B back later.

Table 1(c) demonstrates this point. It shows the discounted resource allocation for each client over time by discounting the possessed extra unused resource. At time t_1, A yields her 30 unused resources to B and B's discounted resources are $65(= 50+30\cdot 50\%)$ instead of $80(= 50 + 30)$. Similarly for A at t_3, it preempts 30 resources from B and its discounted resources are $65(= 50 + 30 \cdot 50\%)$. Still, both of them are *fair* at time t_4.

	Client A					Client 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.

	Client A					Client 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*.

	Client A					Client B				
	Demand		Counted Allocation		Preempt	Demand		Counted Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	−30	100	100	65	65	+30
t_2	40	40	40	60	−10	60	80	55	120	+10
t_3	80	80	65	125	+30	50	70	20	140	−30
t_4	60	60	55	**180**	+10	50	100	40	**180**	−10

(c) Counted allocation results under *discount*-based approach of *LTRF*. There is a discount (e.g., 50%) for the extra unused resources, to incentivize clients to preempt resources actively for system utilization maximization. In this example, although the *counted* allocations for A and B are 180, their real allocations are both 200, which is the same as Table 1(b).

Table 1: A comparison example of *MemoryLess Resource Fairness (MLRF)* and *Long-Term Resource Fairness (LTRF)* in a shared computing system consisting of 100 computing resources for two users A and B.

4.1 Property Analysis for LTRF

THEOREM 1. *LTRF satisfies the sharing incentive property.*

Algorithm 1 LTRF pseudo-code.

1: R**:** total resources available in the system.
2: $\ddot{R} = (\ddot{R}_1, ..., \ddot{R}_n)$**:** current allocated resources. $\ddot{R}_i$ denotes the current allocated resources for client i.
3: $U = (u_1, ..., u_n)$**:** total used resources, initially 0. u_i denotes the total resource consumed by client i.
4: $W = (w_1, ..., w_n)$**:** weighted share. w_i denotes the weight for client i.

5: **while** there are pending tasks **do**
6: **Choose** client i with the smallest total weighted resources of u_i/w_i.
7: $d_i \leftarrow$ the next task resource demand for client i.
8: **if** $\ddot{R} + d_i \leqslant R$ **then**
9: $\ddot{R}_i \leftarrow \ddot{R}_i + d_i$. ▷ Update current allocated resources./*Section 5.2.2*/
10: Update the total resource usage u_i for client i. ▷ /*Section 5.2.2*/
11: Allocate resource to client i. ▷ /*Section 5.2.3*/
12: **else** ▷ The system is fully utilized.
13: **Wait** until there is a released resource r_i from client i.
14: $\ddot{R}_i \leftarrow \ddot{R}_i - r_i$. ▷ Update current allocated resources/*Section 5.2.2*/

PROOF. Consider a shared pay-as-you-use computing system of R resources group-bought by n clients with equal share (or monetary cost) over t period time. When pursuing individually with the same amount of money, 1). the amount of resources R_1 a client can receive is less than $\frac{R}{n}$, as group-buying has discount over personal buying; 2). Under R_1 resources, she can get at most $t \cdot R_1$ resources, smaller than $t \cdot \frac{R}{n}$. In contrast, with group-buying and fair allocation with LTRF, a client can get at least $t \cdot \frac{R}{n}$ resources. Thus LTRF satisfies sharing incentive property. □

THEOREM 2. *(Non-Trivial-Workload Incentive) Any client who submits non-trivial workloads to the shared pay-as-you-use computing system could get benefits under LTRF.*

PROOF. Recall that LTRF focuses on the fairness over total resources with *lending agreement*. When a client's resource demand is less than its current share, she can *lend* unneeded resources out. Later when she needs more resources in the future, she can get extra amount of resources back from others that she lent before. Reversely, if she submits lots of dirty (or trivial) workloads to the system when her *true* demand is less than her share, she will loose opportunity to get more extra sources, especially when she has lots of important and urgent workloads to compute later. Hence, LTRF meets non-trivial-workload incentive property. □

THEOREM 3. *LTRF achieves resource-as-you-pay fairness in a group-buying shared computing system.*

PROOF. Each client in a *shared* computing system has right to enjoy at least the amount of resources that she pays. One key factor that affects resource-as-you-pay fairness is the varied client's demands at different time (i.e., *unbalanced* workload which can be either less or larger than her current share). LTRF overcomes the unbalanced workload problem by considering the fairness at the level of total allocated resources and following *lending agreement*. It adjusts the current allocation of resources to each client dynamically according to her historical total allocated resources and current demand, ensuring that the total resources a client received are *fair* with each other. Thus, LTRF is resource-as-you-pay fairness. □

THEOREM 4. *LTRF satisfies strategy-proofness property.*

PROOF. Theorem 2 has demonstrated that LTRF satisfies non-trivial-workload incentive property that can make a client be *truly* willing to yield out her unused resources when she does not need. On the other hand, it is possible that an overloaded client lies about her true demands to let her get more allocated resources in preemption with others at a time. Due to *lending agreement* requirement under LTRF, the consequence of lying is a pre-overconsumption of her resources and she needs to *pay back* at a later time to others. Thus, lying cannot benefit her at all. □

THEOREM 5. *LTRF satisfies pareto efficiency property.*

PROOF. Recall in our LTRF algorithm, we propose a *discount*-based approach to incentivize users to preempt extra unused resources from others. It indicates that the utilization of system is fully maximized whenever there are pending tasks. Therefore, it is impossible for a client to get more resources without decreasing the resources of others. □

Finally, Table 2 summarizes the properties that are satisfied by MLRF and LTRF, respectively. MLRF is *not* suitable for pay-as-you-use computing system due to its lack of support for three important desired properties, whereas LTRF can achieve all those properties.

Property	Allocation Policy	
	MLRF	LTRF
Sharing Incentive	√	√
Non-Trivial Workload Incentive		√
Resource-as-you-pay Fairness		√
Strategy-Proofness		√
Pareto Efficiency	√	√

Table 2: List of properties for MLRF and LTRF.

4.2 Fairness Definition

Due to the varied resource demands and resource preemption in the shared environment, the total resources a client obtained are undermined. Generally, every client wants to get more resources or at least the same amount of resources in a shared computing system than exclusively using the system. We call it *fair* for a client (i.e., sharing benefit) when that can be achieved. In contrast, it is also possible for the total resources a client received are less than that without sharing, which we call *unfair* (i.e., sharing loss). To ensure resource-as-you-pay 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*.

Without mention, we refer to the *total* resources as *accumulated* resources below. Let $g_i(t)$ be the currently allocated resources for the ith client at time t. Let $f_i(t)$ denote the *accumulated* resources for the ith client at time t. Thus,

$$f_i(t) = \int_0^t g_i(t)\, dt. \quad (1)$$

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

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

The fairness degree $\rho_i(t)$ for the ith client at time t is defined as follows:

$$\rho_i(t) = \frac{\int_0^t g_i(t)\, dt}{\int_0^t \min\{d_i(t), S_i(t)\}}. \quad (3)$$

$\rho_i(t) \geqslant 1$ implies the absolute resource fairness for the ith client at time t. In contrast, $\rho_i(t) < 1$ indicates *unfair*. For a client 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. 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*. Let $\Psi(t)$ be *sharing benefit degree*, as a sum of all $(\rho_i(t) - 1)$ subject to $\rho_i(t) \geqslant 1$, i.e.,

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

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.,

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

We can use this two metrics to compare the quality for different fair policies. Thereby, it always holds that $\Psi(t) \geqslant 0 \geqslant \Omega(t)$. Moreover, 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 pay-as-you-use computing system, either of them could be nonzero. For a good fair policy, it should be able to maximize $\Omega(t)$ first (e.g., $\Omega(t) \to 0$) and next try to maximize $\Psi(t)$.

5. LTYARN: A LONG-TERM YARN FAIR SCHEDULER

YARN is an emerging resource management and job processing system, and has been viewed as a distributed operating system. As a case study, we implement LTRF on YARN. We propose a long-term YARN fair scheduler called *LTYARN*, by generalizing the default instant max-min fairness.

5.1 Long-Term Max-Min Fairness

We present our long-term max-min fairness model for LTYARN.

5.1.1 Challenges and Approaches

Our long-term max-min fairness policy is based on the *accumulated* resources. When estimating the *accumulated* resources for a task, we need to know the capacity and demand of its requested resources and the execution time that it takes. However, there are several challenges for *online* applications (i.e., refers to applications that arrive over time) on that as follows,

1. the execution time of tasks for each application are often different and unknown in advance.
2. the arriving time for each application can be arbitrary and unknown in advance.
3. the computing resources (e.g., CPU powers) can be heterogeneous in a heterogeneous cluster, and the resource demand (e.g., memory size) for each task can be different.

To deal with the above mentioned challenging issues, we provide several methods below,

Time Quantum-based Approach. It is an approximation approach to deal with the first challenging problem. It gives a concept of *assumed execution time*, initialized with a time quantum, to represent the prior unknown *real* execution time. The assumed execution time is adjusted dynamically to make it close to the real execution time.

The details of our approach are that, 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.

Wall Clock-based Approach. It concerns with the second challenging problem of '*online*' arriving. Different applications may arrive at different time. It would be no longer suitable to use the accumulated *consumed* resources as a measure to control the fair share. The explanation is that, from the system's (e.g., global-level) perspective, in order to improve its resource utilization, it often follows the idiom that '*the early bird gets the worm*' (we call it *Early Bird Privilege* next) to incentivize users to submit their applications as early as possible. To achieve that, one solution is to give a penalty for the late arriving application, by only starting to consider (or memorize) the fair share of resources from its arriving time. Moreover, our fairness model is on the basis of max-min fairness algorithm [21]. Technically, to implement it, there is a need to top-up a resource cost, named as *Pseudo Accumulated Resources (PAR)*, such that the fair scheduler will not favor the late arriving application. Thus, in contrast to *offline* application whose accumulated resources can be directly set to its accumulated *consumed* resources as expressed by Formula (1) implicitly, the accumulated resources for each *online* application should include both its *PAR* and accumulated consumed resources. That is, for the online application, the definition in Formula (1) should be modified as,

$$f_i(t) = \int_0^t g_i(t)\, dt + \phi_i(t). \quad (6)$$

where $\phi_i(t)$ denotes the *PAR watched* at time t by the application i. Moreover, by taking into account the *discount*-based approach for extra unused resources proposed by Algorithm 1 of LTRF in Section 4, we have the currently *discounted* allocated resource $g_i^{'}(t)$ as follows:

$$g_i^{'}(t) = \min\{g_i(t), S_i(t)\} + \max\{g_i(t) - S_i(t), 0\} \cdot \eta. \quad (7)$$

where $\eta (0 \leqslant \eta \leqslant 1)$ denotes the discount rate. Hence, the definition of Formula (6) should be further modified as,

$$f_i(t) = \int_0^t g_i^{'}(t)\, dt + \phi_i(t). \quad (8)$$

We call this method *Wall Clock-based Approach*, where the *Wall Clock* refers to a time period before the arriving of an application, as illustrated in Figure 1 (a).

Weighted Resource based Approach. It targets at the third challenge. We assign a *weight* to each heterogeneous resource in terms of its computing capacity. For example, the CPU resource can be weighted based on its clock frequency. Thereby, for the ith application,

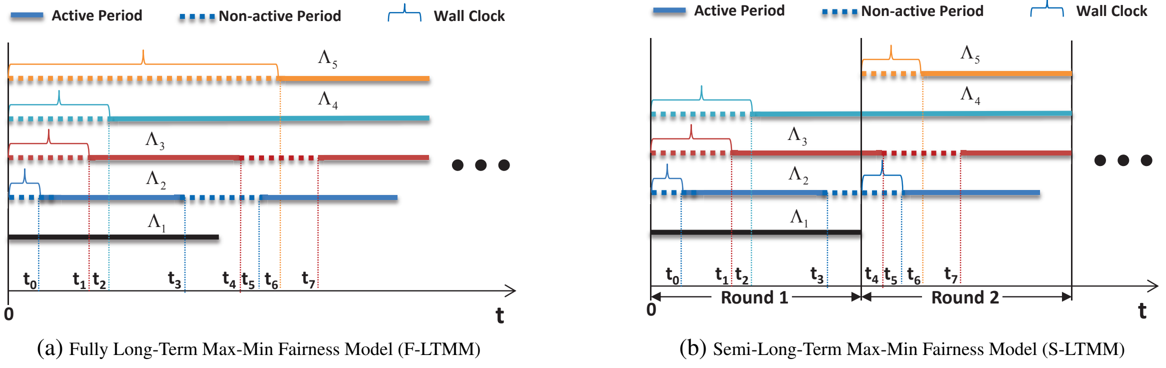
$$g_i(t) = \sum_{j \in \tau_i(t)} \theta_{i,j} \cdot \delta_{i,j} \cdot \alpha_{i,j}(t). \quad (9)$$

where $\tau_i(t)$ denotes the set of tasks from the ith application that are allocated with resources at the time t. $\theta_{i,j}$ and $\delta_{i,j}$ denote the resource demand (e.g., the size of vcore or memory) and weight for the jth task of the ith application, respectively. $\alpha_{i,j}(t)$ represents the *assumed execution time* for the jth task of the ith application at time t. It is our future work to extend the definition to other hardware resources like GPUs [14].

5.1.2 Long-Term Max-Min Fairness Model

This subsection proposes long-term max-min fairness model for LTYARN. YARN is a hierarchical tree structure of multi-level fairness: applications at the bottom and queues at the higher level. We apply the same mechanism for different levels. The following design considers the bottom-level (i.e., application-level).

Let $\Lambda = \{\Lambda_1, \Lambda_2, \Lambda_3, ...\}$ 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 a_i be the arriving time for the application Λ_i. According to the *Early Bird Privilege* and max-min fairness policy, the *PAR* $\phi_i(t)$ for the active application Λ_i should be,



(a) Fully Long-Term Max-Min Fairness Model (F-LTMM)

(b) Semi-Long-Term Max-Min Fairness Model (S-LTMM)

Figure 1: The long-term max-min fairness models for *LTYARN*. For an application, *Active Period* refers to the time interval when it has pending/running tasks available. Otherwise, it belongs to *Non-active Period*. *Wall Clock* refers to a time period before the arriving of an application with respect to the starting time of the current round.

$$\phi_i(t) = \begin{cases} \max\limits_{\Lambda_k \in \tilde{\Lambda}} \{f_k(t) | a_k < a_i\} = \\ \max\limits_{\Lambda_k \in \tilde{\Lambda}} \left\{ \int_0^t g_k'(t)\, dt + \phi_k(t) | a_k < a_i \right\}, & (a_i > \min\limits_{\Lambda_k \in \tilde{\Lambda}} \{a_k\}). \\ 0, & \text{others}. \end{cases} \quad (10)$$

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 (6), (9), (10), the application Λ_i to be chosen at time t for fair resource allocation should satisfy the following condition,

$$\frac{f_i(t)}{\omega_i} = \min_{\Lambda_k \in \tilde{\Lambda}} \left\{ \frac{f_k(t)}{\omega_k} | n_i^p(t) > 0 \right\}. \quad (11)$$

We name this fairness model ***Fully Long-Term Max-Min Fairness Model (F-LTMM)***, as illustrated in Figure 1(a), considering that it is *recording* the consumed resources all the way since YARN system starts working.

In practice, we may not want the system to be fully long-term. Instead, the definition can be applied to a period of time (e.g., 24 hours). It motives us further to propose a time window-based long-term fairness model below.

Semi-Long-Term Max-Min Fairness Model (S-LTMM). The key idea is that, instead of fully memorizing resources all the time since the system starts working, we can divide system working time into a set of time windows (by default, we call the *time window* as *round*). Within the round (i.e., *Intra-Round Phase*), we adopt the fully long-term fairness model. When the system moves to the next round (i.e., *Inter-Round Phase*), it ignores all jobs' history information from the previous round and starts memorizing from the beginning. It is a hybrid of fully long-term fairness model at intra-round phase and memoryless fairness model at inter-round phase.

Figure 1(b) illustrates the model. Let L denote the time length of a computation round, and t^s be the start time of the current computation round. Then we can compute t^s with the following formula,

$$t^s = \begin{cases} t^s + \lfloor \frac{t - t^s}{L} \rfloor \cdot L, & (t > 0). \\ 0, & (t = 0). \end{cases} \quad (12)$$

Moreover, all of the *F-LTMM*-related elements, including *Wall Clock*, *PAR* and accumulated consumed resources for each application, should be updated and counted from t^s instead. Then Formula (6) should be updated to be,

$$f_i(t) = \int_{t^s}^{t} g_i'(t)\, dt + \phi_i(t). \quad (13)$$

Unlike *F-LTMM* whose *Wall Clock* is just equal to the application's arriving time, the *Wall Clock* in *S-LTMM* is round-based, referring to a non-active period of an application since t^s, e.g., Λ_2 in Figure 1(b). We define *Round Arriving Time* $\breve{a}_i$ for Λ_i to be the starting time point at which the application becomes active since t^s, e.g., t_5 for Λ_2 at Round 2 in Figure 1(b). It can be computed based on the following formula,

$$\breve{a}_i = \begin{cases} a_i, & (t^s \leqslant a_i). \\ t^s, & (\exists j \in \tau_i(t), t_{i,j}^s \leqslant t^s < t_{i,j}^c). \\ \min\limits_{j \in \tau_i(t)} \{t_{i,j}^s | t_{i,j}^s > t^s\}, & \text{others}. \end{cases} \quad (14)$$

Let $t_{i,j}^s$, $t_{i,j}^c$ denote the start time and finished time for the jth task of the application Λ_i, respectively. Particularly, for the finished tasks of each application in *S-LTMM*, only the jth task satisfying $t_{i,j}^c > t^s$ will count. According to the time quantum-based approach, we then have,

$$\alpha_{i,j}(t) = \begin{cases} t_{i,j}^c - \max\{t^s, t_{i,j}^s\}, & (t^s < t_{i,j}^c \leqslant t). \\ \max\{Q, t - \max\{t^s, t_{i,j}^s\}\}, & (t < t_{i,j}^c \leqslant t^s + L). \\ 0, & \text{others}. \end{cases} \quad (15)$$

where Q denotes the time quantum. And accordingly, Formula (10) should be updated to

$$\phi_i(t) = \begin{cases} \max\limits_{\Lambda_k \in \tilde{\Lambda}} \left\{ \int_{t^s}^t g_k'(t)\, dt + \phi_k(t) | \breve{a}_k < \breve{a}_i \right\}, & (\breve{a}_i > \min\limits_{\Lambda_k \in \tilde{\Lambda}} \{\breve{a}_k\}). \\ 0, & \text{others}. \end{cases} \quad (16)$$

Finally, by combining Formula (12), (15), (9), (16), (13), similar to *F-LTMM*, we can obtain *S-LTMM* by allocating resources to the application Λ_i subject to Formula (9) stringently at time t.

5.2 Design and Implementation of LTYARN

In YARN, the resources are organized into multiple queues with hierarchical tree structure. Each queue can represent an organization and the resources are shared among them. Figure 3 shows an example of three-level structure. 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.

Figure 2 gives an overview on the design and implementation of LTYARN. It consists of three key components: *Quantum Updater (QU)*, *Resource Controller (RC)*, and *Resource Allocator (RA)*. QU is responsible for updating the time quantum for each queue dynamically. 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. In the following, we present some implementation details about each component.

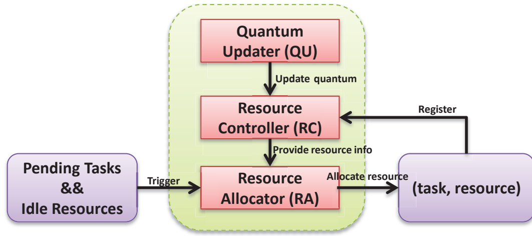


Figure 2: Overview of LTYARN.

5.2.1 Quantum Updater (QU)

For LTYARN, the suitable value of the time quantum Q is very important for *fairness convergency*, which refers to the convergency of unfair applications for their long-term resources at a time point and after that they fairly share the resources with each other. To achieve fast convergency, we need to make Q be close to the real execution time of tasks. Ideally, we need to adapt Q to different applications/tasks and also varied types of applications in different queues for YARN in practice, ensuring that each queue owns a suitable Q for its own applications so that they do not interfere with each other.

We propose an adaptive task quantum policy. It is a multi-level self-tunning approach by extending the hierarchical structure of YARN's resource organization, as shown in Figure 3. 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 flows are a self-tunning procedure, refreshing periodically by a small 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 the initialization process from the top to down. First, it will check whether its parent queue is new one or not (Arrow (1) in Figure 3). If yes, it will assign the root-queue quantum to its parent-queue quantum, e.g., $Q_{1,1} \leftarrow Q_0$. Next, it checks its sub-queues (e.g., leaf-queue) (Arrow (2) in Figure 3). If it is a new one, it will assign its parent-queue quantum to its sub-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 Figure 3).

QU checks the system periodically for new completed tasks. When there is a task finished, the self-adjustment process performs from the bottom to up. First, it will update the time quantum for applications with the average task completion time (Arrow (4) in Figure 3). Next, it updates its leaf-queue quantum with its average application quantum (Arrow (5) in Figure 3). Similarly, it updates its parent-queue quantum using the average value of its leaf-queue quantum (Arrow (6) in Figure 3). Finally, the root-queue quantum is updated with the average value of parent-queue quantum (Arrow (7) in Figure 3).

5.2.2 Resource Controller (RC)

Resource Controller (RC) is the main component of LTYARN. Its principle responsibility is to manage and update the accumulated resources for each queue, needed by RA, on the basis of the model S-LTMM. 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 periodically (e.g., 1 second). In the updating procedure, it first updates the starting time of the current round based on Formula (12) and the round arriving time for each application based on Formula (14). Next, based on time quantum-based approach, it estimates the assumed execution time for each running/completed task with the updated quantum

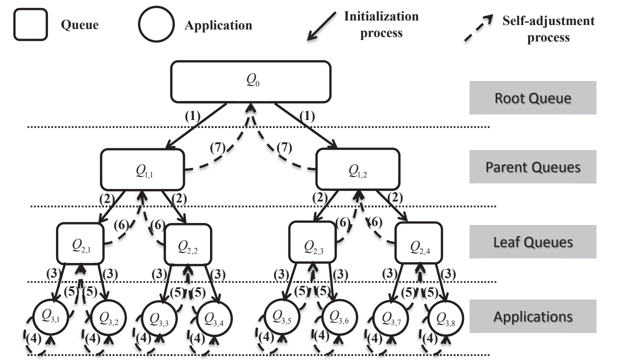


Figure 3: The adaptive task quantum policy for YARN. The *up-to-bottom* data flow is a task time quantum initialization process for new applications. The *bottom-to-up* data flow is a quantum self-adjustment process for existing applications/queues.

value from QU, according to Formula (15). The currently allocated resource for each task can then be estimated with Formula (7). After that, it estimates the Pseduo Accumulated Resources (PAR) for each application based on Formula (16). Finally, it updates the accumulated resource for each application/queue based on Formula (13).

5.2.3 Resource Allocator (RA)

Resource Allocator (RA) is responsible for resource allocation at each queue of different levels, as shown in Figure 3. It is triggered whenever there are pending tasks or 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 (11). 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.

In practice, it is better for its root queue to use the long-term max-min fairness, viewing each of its sub-queues as a client or an organization to it. We need to guarantee the resource-as-you-pay fairness for them. For each parent-queue representing an organization, we should also adopt the long-term max-min fairness if its subqueues (i.e., members of the organization) require resource-as-you-pay fairness. In contrast, when a queue belongs to a client, there might be no need to ensure resource-as-you-pay fairness for its sub-queues. In that case, we can choose either memoryless max-min fairness, long-term max-min fairness or FIFO.

6. EVALUATION

We ran our experiments in a cluster consisting of 10 compute nodes, each with two Intel X5675 CPUs (6 CPU cores per CPU with 3.07 GHz), 24GB DDR3 memory and 56GB hard disks. The latest version of YARN-2.2.0 is chosen in our experiment, used with a two-level hierarchy. The first level denotes the root queue (containing 1 master node, and 9 slave nodes). For each slave node, we configure its total memory resources with 24GB. The second level denotes the applications (i.e., workloads).

6.1 Macro-benchmarks

We ran a macro-benchmark consisting of four different workloads. Thus, four different queues are configured in YARN/LTYARN,

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

Table 3: Job types and sizes for each bin in our synthetic Facebook workloads.

namely, *Facebook*, *Purdue*, *Spark*, *HIVE/TPC-H*, corresponding to the following workloads, respectively. 1). A MapReduce instance with a mix of small and large jobs based on the workload at Facebook. 2). A MapReduce instance running a set of large-sized batch jobs generated with Purdue MapReduce Benchmarks Suite [1]. 3). Hive [24] running a series of TPC-H queries. 4). Spark [32] running a series of machine learning applications.

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. [33]. The workload consists of 100 jobs. We categorize them into 9 bins of job types and sizes, as listed in Table 3. It is a mix of large number of small-sized jobs ($1 \sim 15$ tasks) and small number of large-sized jobs (e.g., 800 tasks[2]). The job submission time is derived from one of SWIM's Facebook workload traces (e.g., FB-2009_samples_24_times_1hr_1.tsv) [12]. The jobs are from Hive benchmark [5], 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., WordCount, TeraSort, Grep, InvertedIndex, HistogramMovices) randomly from Purdue MapReduce Benchmarks Suite [1]. We use 40G wikipedia data [26] for WordCount, InvertedIndex 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.

Hive / TPC-H. To emulate continuous analytic query, such as analysis of users' behavior logs, we ran TPC-H benchmark queries on Hive [3]. 40GB data are generated with provided data tools. Four representative queries Q1, Q9, Q12, and Q17 are chosen, each of which we create five instances. We launch one query after the previous one finished in a round robin fashion.

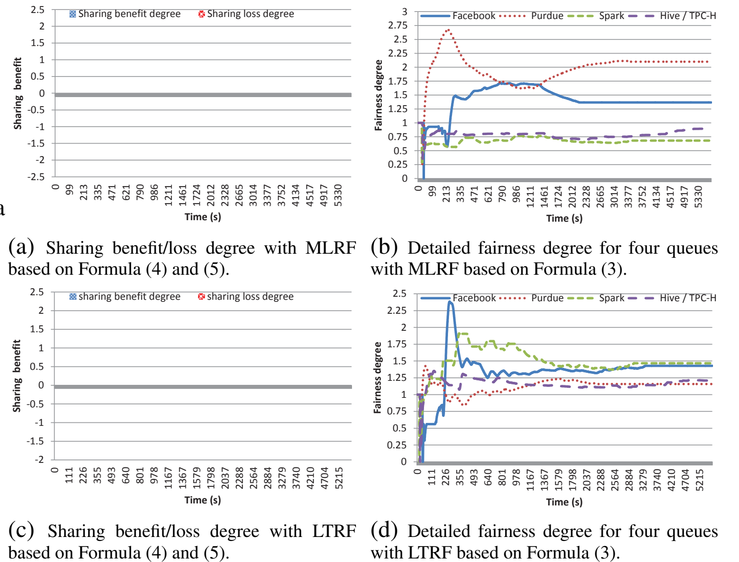
Spark. Latest version of Spark has supported its job to run on the YARN system. We consider two CPU-intensive machine learning algorithms, namely, kmeans and alternating least squares (ALS) with provided example benchmarks. We ran 10 instances of each algorithm, which are launched by a script that waits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

6.2 LTRF Resource Allocation Flow

To understand the dynamic history-based resource allocation mechanism of LTRF under 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 Figure 4. Figure 4(a) and 4(b) show the normalized results of the current resource demand and currently allocated resources for each workload with respect to its current share. Figure 4(c) presents the normalized accumulated resources for four workloads with respect to the system capacity.

Figure 4(a) shows that workloads have different resource demands over time. At the beginning, Purdue, Spark and Hive / TPC-H have an overloaded demand period (e.g., Purdue: $24-131$, Spark: $28-118$, HIVE / TPC-H: $28-146$). Figure 4(b) shows the allocation details for each workload over time. During the common overloaded period of $28-118$, the curves for Purdue, Spark and Hive / TPC-H are fluctuated, indicating that LTRF 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 curves are close to each other) during the period $80-118$, as shown in Figure 4(c). However, for Facebook workload, its overloaded period occurs from $204-260$. During this period, the Purdue workload is also overloaded, as shown in Figure 4(a). To achieve the accumulated resource fairness, LTRF allocated a large amount of resource to it (e.g., $3.85/4.0 = 96.25\%$ at point 222) shown in Figure 4(b), to make it catch up with others. As in the accumulated resource results in Figure 4(c) that, during $204-260$, there is a significant increment for Facebook workload, whereas other workloads increase slightly.

6.3 Macrobenchmark Fairness Results



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

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

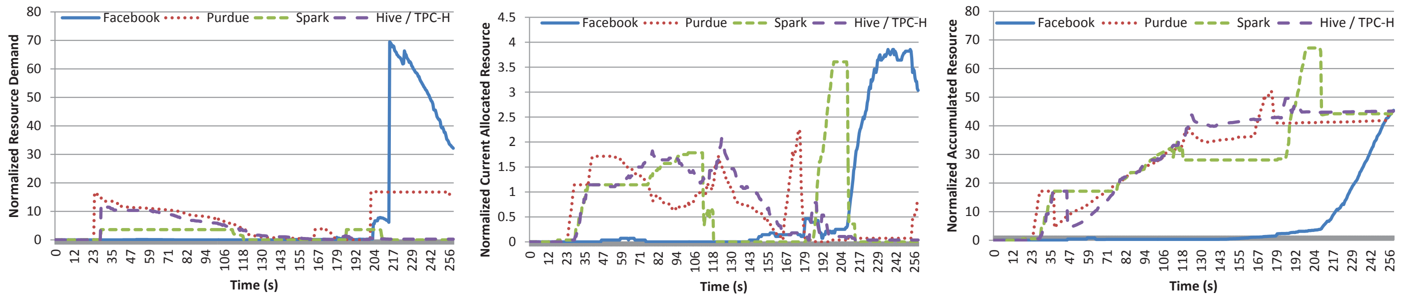
(c) Sharing benefit/loss degree with LTRF based on Formula (4) and (5).

(d) Detailed fairness degree for four queues with LTRF based on Formula (3).

Figure 5: Comparison of fairness results over time for each of workloads under MLRF and LTRF in YARN. All results are relative to the static partition scenario (i.e., non-shared case) 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.

In Section 4.2, we have shown that a good sharing policy should be able to first minimize the sharing loss, and then maximize the sharing benefit as much as possible (i.e., Sharing incentive). We make a comparison between MLRF and LTRF for four workloads over time in Figure 5. All results are relative to the static partition case (without sharing) with fairness degree of one and sharing benefit/loss degrees of zero. Figures 5(a) and 5(c) present the sharing benefit/loss degrees based on Formulas (4) and (5), respectively, for MLRF and LTRF. Figures 5(b) and 5(d) show the detailed fairness

[2] We reduce the size of the largest jobs in [33] 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.

Figure 4: Overview of detailed fairness resource allocation flow for LTRF.

degree for each queue (workload) over time. We have the following observations:

First, the sharing policies of both MLRF and LTRF can bring sharing benefits for queues (workloads). For example, both Facebook and Purdue workloads, illustrated in Figure 5(b) and 5(d) 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, LTRF has a much better result than MLRF. Specifically, Figure 5(a) indicates that the sharing loss problem for MLRF is constantly available until all the workloads complete (e.g., ≈ -0.5 on average), contributed primarily by Spark and TPC-H workloads given by Figure 5(b). In contrast, there is no more sharing loss problem after 650 seconds for LTRF, i.e., all workloads get sharing benefits after that. The major reason is that MLRF 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 Figure 5(b)); 2). some workloads obtain much less resources that without sharing over time (e.g., Spark and TPC-H workloads in Figure 5(b)). In contrast, LTRF 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 and *lending agreement* so that each queue can obtain a much closer amount of total resources over time.

Finally, regarding the sharing loss problem at the early stage (e.g., $0 \sim 650$ seconds) of LTRF in Figure 5(c), it is mainly due to the unavoidable waiting allocation problem at the starting stage, i.e., 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. The problem exists in both MLRF and LTRF. Still, LTRF can smooth this problem until it disappears over time via *lending agreement*, while MLRF cannot.

6.4 Macrobenchmark Performance Results

Figure 6 presents the performance results (i.e., speedup) for four workloads under Static Partitioning, MLRF and LTRF, respectively. All results are normalized with respect to Static Partitioning (i.e., non-shared executions). We see that, 1). the shared cases (i.e., MLRF and LTRF) can possibly achieve better performance than or at least the same as the non-shared case. For example, for Facebook and Purdue workloads, both MLRF and LTRF have much better performance results (e.g., $14\% \sim 19\%$ improvement for MLRF, and $10\% \sim 23\%$ for LTRF) than exclusively using a static partitioning system. The finding is consistent with previous works such as Mesos [15]. The performance gain is mainly due to the resource preemption of unneeded resources from other queues in a shared

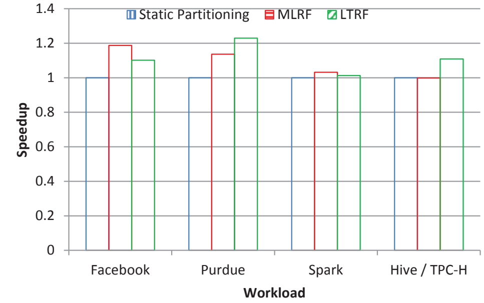


Figure 6: The normalized performance results (e.g., speedup)for Static Partitioning, MLRF and LTRF, with respect to Static Partitioning.

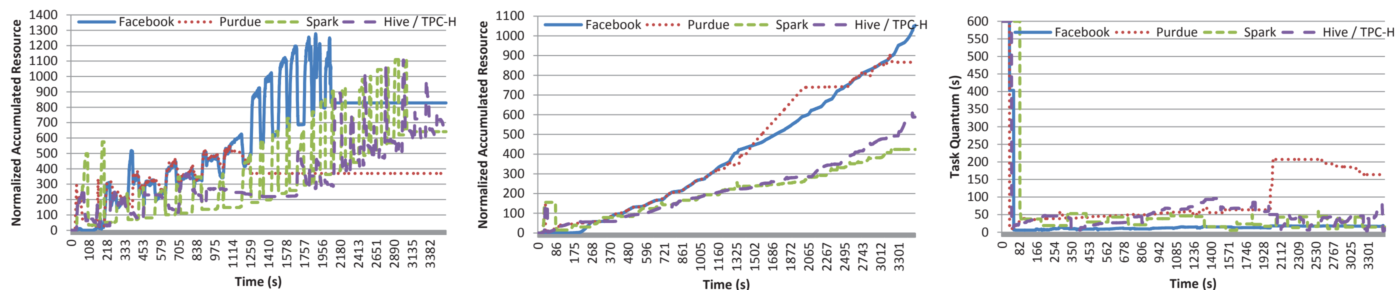
system. The statement can also be validated in Figure 5(b) and 5(d) in Section 6.3. The fairness degrees for both Facebook and Purdue workloads are above one (i.e., get sharing benefit) during the most of time. 2). There is no conclusive result regarding which one is absolutely better than the other between MLRF and LTRF. For example, MLRF is better than LTRF for Facebook by about 7% and Spark by about 2%. However, LTRF outperforms MLRF for Purdue workload by about 8% and TPC-H by about 10%.

6.5 Adaptive Task Quantum Policy Evaluation

To demonstrate the importance and effectiveness of adaptive task quantum policy for YARN, we study the effects of accumulated resource results over time under the fixed time quantum and the adaptive task quantum mechanism proposed in Section 5.2.1.

We consider a scenario where the configured task quantum (e.g., 600s) is much larger than the real task execution time of workloads. Figure 7 shows the compared accumulated results for LTRF over time within one hour, which are normalized with respect to the system capacity. We have the following observations:

First, Figure 7(a) illustrates that the accumulated resource under the fixed task time quantum policy fluctuates significantly over time, making it unable to be an indicator for resource-as-you-pay fairness. This is due to the computation method for assumed execution time in the time quantum-based approach: 1). the assumed execution time for the completed task is equal to its real execution time; 2). for the running task, we compute its assumed execution time using the maximum value of the configured time quantum and its real execution time. Take Facebook workload as an example. Its average task execution time is about 11s. At time 1439s, there are 107 running tasks, whose assumed execution time is 600, and its normalized accumulated resource is 1019. However, at time 1450s (i.e., after 11s), there are 31 running tasks, indicating that at least 76 tasks completed during this period and a significant drop occurs for its normalized accumulated resource (e.g., 630).



(a) Normalized accumulated resources under the fixed task time quantum of 600s, with respect to the system capacity.

(b) Normalized accumulated resources with adaptive task quantum mechanism, with respect to the system capacity.

(c) Adaptive task quantum, initially 600s.

Figure 7: The adaptive task quantum results for LTRF in one hour.

In contrast, with adaptive task quantum policy, as shown in Figure 7(b), the curves of accumulated resource become much smoother, making it good as an indicator for resource-as-you-pay fairness. Figure 7(c) shows the adaptive task quantum results over time for four workloads. We see that each workload has varied task quantum and our policy can adjust them dynamically for all the workloads, validating the effectiveness of our adaptive approach.

7. CONCLUSION AND FUTURE WORK

Pay-as-you-use computing systems have been become emerging in data centers and supercomputers. Resource fairness is an important consideration for such shared environments. However, this paper finds that, the classical *memoryless* resource fairness policies, widely used in many existing popular frameworks and schedulers, including Hadoop, YARN, Mesos, Choosy, Quincy, DHFS [27], MROrder [28], are *not* suitable in pay-as-you-use computing system due to three serious problems, i.e., trivial workload problem, strategy-proofness problem and resource-as-you-pay problem. To address these problems, we propose LTRF and demonstrate that it is *suitable* for pay-as-you-use computing system. Besides, we also propose five payment-oriented properties as metrics to measure the quality for any fair policy in a pay-as-you-use computing system. We developed *LTYARN*, a long-term max-min fair scheduler for the latest version of YARN and our experiments demonstrate the effectiveness of our approaches. As future work, we plan to extend our fairness definition to different price schemes [30] and multiple resource types (such as DRF [11]).

The implementation of LTYARN can be found in http://sourceforge .net/projects/ltyarn/.

8. ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments. Bingsheng He was partly supported by a startup Grant of Nanyang Technological University, Singapore.

9. REFERENCES

[1] F. Ahmad, S. Y. Lee, M. Thottethodi, T. N. Vijaykumar. *PUMA: Purdue MapReduce Benchmarks Suite*. ECE Technical Reports, 2012.

[2] Apache. *YARN*. https://hadoop.apache.org/docs/current2/index.html

[3] Apache. *TPC-H Benchmark on Hive*. https://issues.apache.org/jira/browse/HIVE-600.

[4] Apache. *Hadoop*. http://hadoop.apache.org.

[5] Apache. *Hive performance benchmarks*. https://issues.apache.org/jira/browse/HIVE-396.

[6] H. Arabnejad, J. Barbosa. *Fairness Resource Sharing for Dynamic Workflow Scheduling on Heterogeneous Systems*, ISPA, pp. 633-639, 2012.

[7] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, I. Stoica. *Hierarchical Scheduling for Diverse Datacenter Workloads*. SOCC'14, 2014.

[8] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04, 2004.

[9] A. Demers, S. Keshav, S. Shenker. *Analysis and Simulation of a Fair Queuing Algorithm*. In SIGCOMM'89, pp. 1-12, 1989.

[10] A. Ghodsi, M. Zaharia, S. Shenker and I. Stoica. *Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints*, EuroSys 2013, April 2013.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Schenker,I. Stoica. *Dominant Resource Fairness: Fair Allocation of Multiple Resource Types*. In NSDI'11, pp. 24-37, 2011.

[12] GitHub. *Facebook workload traces*. https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[13] Group Buying. http://en.wikipedia.org/wiki/Group_buying.

[14] B.S. He, W.B. Fang, Q. Luo, N.K. Govindaraju, T.Y. Wang. *Mars: A MapReduce Framework on Graphics Processors*, In PACT'08, pp.260-269, 2008.

[15] 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*, NSDI 2011, March 2011.

[16] M. Isard, M. Budiu, Y. Yu, A. Birell, D. Fetterly. *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*. Eurosys'07, pp.59-72, 2007.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. *Quincy: Fair Scheduling for Distributed Computing Clusters*, In SOSP'09, pp 261-276, 2009.

[18] R. Jain, D. M. Chiu, and W. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Technical Report EC-TR-301, 1984.

[19] I. Kash, A. D. Procaccia, N. Shah. *No agent left behind: dynamic fair division of multiple resources*. In AAMAS'13, PP. 351-358, 2013.

[20] Loan agreement. http://en.wikipedia.org/wiki/Loan_agreement.

[21] Max-Min Fairness (Wikipedia). http://en.wikipedia.org/wiki/Max-min_fairness.

[22] J. Ngubiri, M. V. Vliet. *A Metric of Fairness for Parallel Job Schedulers*. Journal of Concurrency and Computation: Practice & Experience. Vol 21, PP. 1525-1546, 2009.

[23] G. Sabin, G. Kochhar, P. Sadayappan. *Job Fairness in Non-Preemptive Job Scheduling*. ICPP, pp. 186-194, 2004.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu. *Hive- A Petabyte Scale Data Warehouse Using Hadoop*. ICDE, pp. 996-1005, 2010.

[25] D. C. Parkes, A. D. Procaccia, N. Shah. *Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities*. In ACM Conference on Electronic Commerce, pp. 808-825, 2012.

[26] PUMA Datasets. http://web.ics.purdue.edu/f̃ahmad/benchmarks/datasets .htm.

[27] S.J. Tang, B.S. Lee, B.S. He. *Dynamic slot allocation technique for MapReduce clusters*, In CLUSTER'13, pp. 1-8, 2013.

[28] S.J. Tang, B.S. Lee, B.S. He. *MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads*, In Euro-Par'13, pp. 291-304, 2013.

[29] C.A. Waldspurger, W. E. Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management*. In OSDI'94, 1994.

[30] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, L. Zhou. *Distributed systems meet economics: pricing in the cloud*, In HotCloud'10, pp.1-6, 2010.

[31] W. Wang, B. C. Li, B. Liang. *Dominant Resource Fairness in Cloud Computing Systems with Heterogeneous Servers*. INFOCOM'14, 2014.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. *Spark: Cluster Computing with Working Sets*. HotCloud'10, pp. 10-16. 2010.

[33] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy,S. Schenker,I. Stoica, *Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling*. In Proceedings of EuroSys, pp. 265-278, 2010.

[34] H. N. Zhao, R. Sakellariou. *Scheduling multiple DAGs onto heterogeneous systems*. IPDPS, pp. 159-172, 2006.