Long-Term Multi-Resource Fairness for Pay-as-you Use Computing Systems

Shanjiang Tang, Zhaojie Niu, Bingsheng He, Bu-Sung Lee, and Ce Yu

Abstract—Many current computing systems such as clouds and supercomputers charge users for their resource usages. A user's demand is often changing over time, indicating that it is difficult to keep the high resource utilization all the time for cost efficiency. Resource sharing is a classical and effective approach for high resource utilization. In view of the heterogeneous resource demands of users' workloads, multi-resource allocation fairness is a must for resource sharing in such *pay-as-you-use computing systems*. However, we find that, existing multi-resource fair policies such as Dominant Resource Fairness (DRF), implemented in currently popular resource management systems such as Apache YARN [4] and Mesos [23], are *not* suitable for the pay-as-you-use computing systems. We show that this is because of their *memoryless* characteristic that can cause the following problems in the pay-as-you-use computing systems: 1). users can get resource benefits by cheating; 2). users might not be able to get the total amount of resources that they are entitled to in terms of their resource contributions. In this paper, we propose a new policy called H-MRF, which generalizes DRF and Asset Fairness with the long-term notion. We show that it can address these problems and is suitable for pay-as-you-use computing systems. We have implemented it into YARN by developing a prototype called *MRYARN*. Finally, we evaluate H-MRF using both testbed and simulated experiments. The experimental results show that there are about $1.1 \sim 1.5$ sharing benefit degrees and $1.2\times \sim 1.8\times$ performance improvement for users with H-MRF, better than existing fair schedulers.

Index Terms—Long-term multi-resource fairness, cloud computing, supercomputing, YARN, MRYARN

1 Introduction

Current Clouds (e.g., Amazon EC2, Microsoft Azue) and supercomputers (e.g., *Blue Waters* [5], Comet [6] and *Gordon* [10]) are typically composed of thousands of compute nodes connected via a high-speed network. Both of them are *pay-as-you-use computing systems* that charge users according to the length of their jobs' execution and the number of compute nodes that they request [15]. Various pricing schemes (such as on-demand pricing, reservation, and auction) have been proposed, and new pricing schemes are likely to be introduced in the future [12].

Resource utilization is a key design issue for both users and resources providers [15], [32] to achieve high performance and cost efficiency. However, the fact is that the resource utilization of current computing systems is far from ideal. Delimitrou et al. [17] 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.

- *S. Tang and C. Yu are with the School of Computer Science & Technology, Tianjin University, Tianjin 300350, China. E-mail: {tashj, yuce}@tju.edu.cn.*
- *Z. Niu and B. He are with the School of Computing, National University of Singapore, Singapore 119077. E-mail: nzjemail@gmail.com, hebs@comp.nus.edu.sg.*
- *B.-S. Lee is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: ebslee@ntu.edu.sg.*

Manuscript received 20 June 2017; revised 19 Oct. 2017; accepted 25 Dec. 2017. Date of publication 1 Jan. 2018; date of current version 6 Apr. 2018. (Corresponding author: Shanjiang Tang.)
Recommended for acceptance by Y. Lu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2788880

Regarding the low resource utilization, another observation is that the resource demands of submitted tasks are often *heterogeneous*, as more *diversified* workloads including big data are being deployed and run on the computing systems. In Fig. 1, it shows a resource usage profile of tasks from Google in a data center of 12 thousands of machines based on the Google trace [8] over about a month-long period (May 2011). The position of a circle indicates the CPU and memory resources consumed by tasks. The size of a circle is logarithmic to the number of tasks in the region of the circle. It shows that there are significantly varied demands for tasks on CPU and memory resources, which can cause unbalanced utilization and fragmentation on individual resource types. Thus, to efficiently allocate resources and satisfy heterogeneous resource requirements, we need to consider *multi-resource allocation* that takes multiple resource types into account [20].

Resource sharing is a classical and effective approach for high resource utilization [20]. 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 allowing overloaded users to utilize unused resources from underloaded users. To achieve resource sharing in a pay-as-you-use computing system, users can contribute their requested computing nodes to the resource pool and share with each other by using existing resource management systems such as Mesos and YARN. Fairness is an important system issue in resource sharing [31]. Only when the fairness is guaranteed for users, the resource sharing can be possible in multi-resource allocation. We have

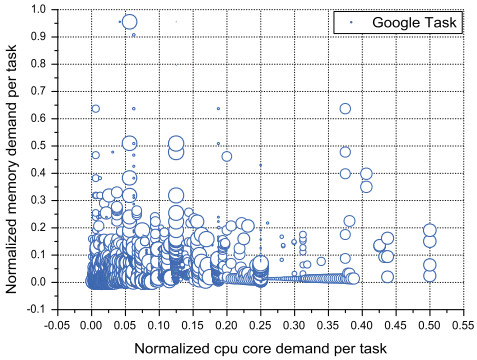


Fig. 1. Heterogeneous resource demands for tasks from google traces [8].

identified the following *good* properties that a fair multi-resource allocation policy in a *shared* pay-as-you-use computing system should satisfy:

- *All users should be better off in shared resource allocation than under an exclusively non-sharing environment. (sharing incentive)*
- *Cheating users should not get resource benefits by misreporting their resource demands. (truthfulness)*
- *It is not possible to increase the allocation of a user without decreasing the allocation of at least one user. (Pareto Efficiency)*
- *The aggregate accumulated resource usage[1] of multi-resource types used by users should be proportional to their resource contributions over time. (resource-as-you-contributed fairness)*

Dominant Resource Fairness (DRF) [20] is one of the most popular multi-resource allocation policies. It has been widely used in existing systems such as Mesos and YARN. DRF was originally designed for a traditional cluster environment where the computing resources are provided by a third-party instead of users [20]. When it comes to the pay-as-you-use computing system, unfortunately, we show that it fails to meet some of the above requirements (See details in Section 3.2), due to its '*memoryless*' allocation feature (i.e., allocating resources fairly at the instant time without considering historical allocations). It indicates that DRF is *not* suitable for pay-as-you-use computing.

Therefore, we investigate a fair resource management policy for the pay-as-you-use computing system. By analyzing problems of existing multi-resource allocation policies, we attempt to explore a multi-resource allocation policy by extending the existing memoryless policies with the long-term notion so that the historical resource allocations are considered. However, enabling multi-resource allocation to satisfy all of the aforementioned requirements is challenging. Particularly, it can be a non-trivial task to define and ensure resource-as-you-contributed fairness in the case of multiple types of resources. As we will show (in Section 5.1), the naive extension of DRF with the long-term notion (Attempt 1 in Section 5.1) cannot achieve the resource-as-you-contributed fairness, since only the dominant resource is considered. We also attempt to extend another popular fairness policy (Asset Fairness) with the long-term notion (Attempt 2 in Section 5.2), where resources of all types are considered. Unfortunately, we show that it also fails to satisfy sharing incentive property.

In this paper, by observing that the two attempts are complementary with each other, we propose a multi-resource allocation policy, called *H-MRF* for the pay-as-you-use computing system, by generalizing DRF and Asset Fairness [20] with the long-term notion. H-MRF ensures that each user in the pay-as-you-use computing system can at least get the amount of total resources as that under the exclusively non-sharing environment in the long-term view. Moreover, H-MRF can guarantee that no users can get a larger amount of total allocated resources over time by lying about their demands.

We have implemented our H-MRF policy in Apache YARN [4], an emerging open-source system infrastructure, and developed a prototype named MRYARN.[2] We have used two complementary methods to evaluate the effectiveness of our proposed approach: real experiments in a Amazon EC2 cluster and trace-driven simulations. The experimental results in the Amazon EC2 cluster show that, MRYARN can achieve sharing benefit for each user in the pay-as-you-use computing system, better than other baseline policies (e.g., DRF and the two attempts). Second, resource sharing with MRYARN can achieve higher resource utilization and better performance than exclusively non-sharing computation. Moreover, MRYARN outperforms other baseline policies in performance due to its efficient task placement in reducing machines' fragmentation. In addition, we also conduct a simulation-based experiment at a large scale with Google cluster-usage traces. The simulation results are consistent with that of MRYARN. Third, users under H-MRF can achieve the *lowest* monetary cost payment compared with other policies (e.g., DRF) in the shared pay-as-you-use computing system.

The rest of the paper is organized as follows. Section 2 presents some desirable allocation properties for pay-as-you-use computing systems. Section 3 introduces the background and gives the motivations for our work. We give a multi-resource fairness definition in Section 4. Section 5 proposes and analyzes our long-term multi-resource fairness polices. Section 6 gives the implementation of H-MRF policy in YARN. The experimental evaluation is given in Section 7. We review the related work in Section 8. Finally, Section 9 concludes the paper.

2 Desirable Allocation Properties

We present several allocation properties that are essential and desirable for resource allocation policies in the *shared* pay-as-you-use computing system, whose compute nodes are contributed by different users.

Sharing Incentive. Each user should be better off sharing resources, than exclusively using her own partition of resources. Only this, users are willing to share their resources with others actively. This property is non-trivial as it is a

1. *Aggregate accumulated resource usage* refers to the aggregation of accumulated resources of multiple resource types over time. For example, assume that one CPU is worth one GB memory (See Section 3.3). If there is a multi-resource allocation of <2 CPUs, 10 GB$>$ lasting for 60 seconds, then the value of aggregate accumulated resource is $2 \times 60 + 10 \times 60 = 720$.

2. The source code of MRYARN: https://sourceforge.net/projects/mryarn/

service-level agreement (SLA) guarantee of resource allocation for users in the pay-as-you-use computing system.

Resource-as-you-Contributed Fairness. In the *shared* pay-as-you-use computing system, assume that each user contributes a certain number of machines (resources) to the common pool of machines (resources) in one period. Then, the aggregate accumulated resource usages of multi-resource types that each user *used* should be proportional to her resource contribution in the shared environment over time. For example, if there is a *shared* 100 nodes computing system for which User A contributes 20 nodes and User B contributes 80 nodes, then the resource-as-you-contributed fairness is guaranteed when the ratio of the aggregate accumulated resource of multi-resource types for User A to B is $\frac{20}{80}$.

Truthfulness. Any user in the system should not be able to get resource benefits by lying about her resource demands. This property is essentially important for a pay-as-you-use computing system, as in real-world systems, users may try to manipulate the schedulers for more allocation by lying about their resource demands [20], [21], [36].

Pareto Efficiency. An allocation policy is pareto efficiency if it is impossible to increase the allocation of a user without decreasing the allocation of at least one user. This property is critical for high resource utilization.

3 Background and Motivation

In this section, we start by reviewing some background of accounting and allocation models on current clouds and supercomputers, and present our work settings. Next we describe existing *memoryless* multi-resource fairness policies, including DRF and Asset Fairness, and motivate our work by showing their problems.

3.1 Accounting Models on Clouds and Supercomputers

For the sake of better understanding why resource-as-you-contributed fairness is so important for pay-as-you-use computing systems, we can take a look at accounting and allocation models in cloud computing and supercomputing, respectively.

Cloud Computing. The cloud resources/services are offered to users on the basis of pay-as-you-use business model. Users pay for compute resources by per second (i.e., billed on one second increments, with a 60 second minimum) or by per hour billing (i.e., less than hour will be automatically rounded to an hour) depending on which instances users run. To meet different users' needs of big data applications, cloud providers generally offer several options of pricing schemes. For example, Amazon EC2 provides users with three pricing schemes, i.e., on-demand pricing, reservation pricing and spot(auction) pricing. For on-demand instance, its price is fixed and a bit higher than that of reservation instance. In contrast, for reservation instance, its cost consists of two parts. One part is the *upfront* cost. The other part is the discounted hourly-based cost. It is always guaranteed to be available whenever a user requests it. For spot instance, its price varies over time, which is often much lower than on-demand and reservation instances. However, it is not always available for requesting.

Supercomputing. Most current supercomputers such as *Blue Waters* [5], Comet [6] and *Gordon* [10] are supported by National Science Foundation (NSF). To utilize current systems, users need first submit grant proposals to NSF for review and then obtain awarded time as a finite number of *Service Unit (SU)*. When a job of a user is running on a system, it consumes the user's bank of SUs at a rate that is proportional to the job' execution time and the number of compute nodes requested by that user [15]. Notably, lots of users' submitted workloads are big data applications (e.g., data-centric climate simulation [38], fluid dynamics simulation [29]), which often take hours or days to finish.

Given those long running applications, to achieve the full cost savings on either of the above two computing systems, users must commit to have a high utilization for compute nodes that they request. In practice, it is most likely that the resource demand of a user is changing over time, implying that it is hard to guarantee the resources to be fully utilized all the time. As we have discussed in Section 1, resource sharing is an effective approach to address it by allowing overloaded users to utilize the unused resources contributed by underloaded users at runtime, and vice versa.

In this paper, let us consider a *shared* computing system whose computing nodes are *contributed* and by n users, where User i makes a contribution of K_i compute nodes (whose pricing schemes can be the same or different across users) to the resource pool. Moreover, we allow users to join in the system at arbitrary time by contributing their computing nodes to the system dynamically. This resource allocation model in fact is similar to existing resource management systems such as YARN. In exchange for the use of their computing nodes, users often want to see something in return. For example, an underloaded user who yields its unused resources to others at the current moment often wants to get more resources back when its demands become large in future. That is, 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 system (i.e., resource-as-you-contributed fairness). In a nutshell, to enable resource sharing sustainable for users in the long run, it is important to explore a fair resource allocation policy for the pay-as-you-use computing system that can meet all the aforementioned *good* properties listed in Section 2.

3.2 Dominant Resource Fairness

Dominant Resource Fairness was recently proposed by [20] for multi-resource allocations and has quickly attracted a remarkable amount of attention [14], [26], [28], [36]. It introduces the concept of a user's *dominant share*, which is the highest share of any typed resource that the user has been allocated. The resource corresponding to the dominant share is called *dominant resource*. Consider an example as follows:

Example 1. Consider a *shared* pay-as-you-use computing system consisting of 100 CPUs and 100 GB memory in total. It is contributed (paid for) by two users A and B equally with the task requirement of <1 CPU, 2 GB$>$ for A, and <1 CPU, 1 GB$>$ for B.

In *Example 1*, User A's dominant resource is memory because each task of A consumes $1/100$ of the total CPUs

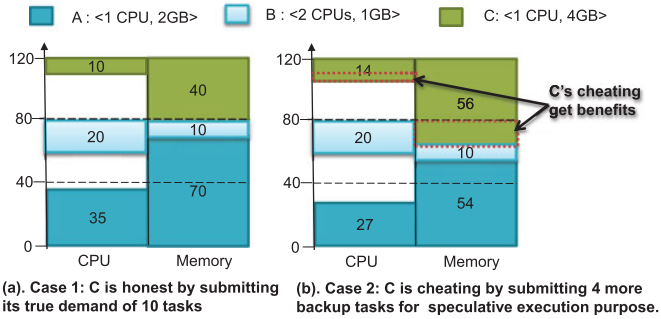


Fig. 2. A counterexample showing that DRF is not truthfulness.

and $2/100$ of the total memory. In contrast, for User B, both CPU and memory are its dominant resources since each task from B consumes $1/100$ of the total CPUs and memory.

DRF achieves the multi-resource fairness by equalizing the dominant shares across all users at the point of time considering resource allocation (i.e., *memoryless*). Consider the previous example of *Example 1*, the resulting allocation will be <25 CPUs, 50 GB$>$ for A and <50 CPUs, 50 GB$>$ for B, making A and B receive the same dominant share of $50/100$. DRF is based on an assumption that equal share ratios of different typed resources are of equivalent value (i.e., 1 percent of all CPUs worth is the same as 1 percent of all memory), since it assumes *fair* when the dominant shares across users are equal, no matter whether the respective dominant resources are of the same type or not.

Problems of DRF. DRF was originally proposed by Ghodsi et al. [20] for a traditional cluster environment of which resources are free for users to utilize. Although there are lots of good properties (e.g., sharing incentive, pareto efficiency), we find that there are several problems for DRF when it is applied to the computing system that charge users for the use of computing resources.

Untruthfulness Problem. A robust policy should have truthfulness property, which can prohibit cheating users from getting benefits. In [20], the authors showed that DRF satisfies the truthfulness property (i.e., strategy-proofness) with an implicit assumption of *unbounded* numbers of tasks for users at any time. However, this does not hold in practice, as the number of tasks per user is generally limited and varying over time. Given a bounded number of tasks for users, we show that DRF *fails* to meet the truthfulness property when there are *more than two users* in the play game through a three-user example as follows.

Example 2. Consider a *shared* pay-as-you-use computing system consisting of 120 CPUs and 120 GB memory, which is contributed (paid by) by three users A, B, C equally. Assume that the task requirements for A, B and C are <1 CPU, 2 GB$>$, <2 CPUs, 1 GB$>$ and <1 CPU, 4 GB$>$, respectively. Moreover, let us suppose that the true numbers of tasks for A, B and C are 50, 10 and 10, respectively. Then the true resource demands for A, B and C are <50 CPUs, 100 GB$>$, <20 CPUs, 10 GB$>$ and <10 CPUs, 40 GB$>$, respectively.

In *Example 2*, the resource share for each of A, B and C is <40 CPUs, 40 GB$>$. Suppose Users A and B are honest users whereas C is an adversarial user. In that case, B has <20 CPUs, 30 GB$>$ unused resources and it yields them to others honestly. However, C can manipulate the scheduler by submitting more than its true number of 10 tasks to compete with A for the unused resources from B. For example, C can submit 4 more backup tasks for the purpose of *speculative execution* [41], which can improve its performance by mitigating the impact of straggled tasks. Based on DRF, the resulting allocation is illustrated in Fig. 2b, where C gets <14 CPUs, 56 GB$>$. In fact, if C is honest, she can only get <10 CPUs, 40 GB$>$, as illustrated in Fig. 2a. Thus, User C gets <4 CPUs, 16 GB$>$ benefits for its speculative execution. Typically, if such a case (e.g., Fig. 2b) often takes place for C, it will allow C to get more benefits over time. Therefore, DRF *violates* truthfulness property in the long-term view.

Regarding this, the behind reason is that there is no penalty for cheating users under DRF policy due to its *memoryless* resource allocation property. Compared with the honest case (Fig. 2a), lying can make users preempt more unused resources (Fig. 2b) from underloaded users with no cost.

Example 3. Consider a *shared* pay-as-you-use computing system with 100 CPUs and 100 GB memory contributed (paid for) by two users A and B equally, where A runs tasks with the demand vector of <1 CPU, 4 GB$>$, and B is with the demand vector of <4 CPUs, 1 GB$>$ per task. Assume that at time t_1, t_2, t_3 and t_4, the numbers of *new* submitting tasks for User A are 30, 20, 27 and 10, and for User B are 4, 24, 8 and 30, respectively.

Resource-as-you-Contributed Unfairness Problem. As a service level agreement, we should ensure that the total resources used by each user are proportional to her contribution (i.e., resource-as-you-contributed fairness). However, due to the fact that a user's resource demands are varying over time, we find that DRF fails to guarantee SLA. Let's illustrate it with Example 3. With DRF, it will be *fair* for A and B when each of them has 20 tasks allocated (i.e., <20 CPUs, 80 GB$>$ resources for A and <80 CPUs, 20 GB$>$ resources for B). The resulting allocation based on DRF is given in Table 1 a. At time t_1, the number of tasks for B is 4. B consumes <16 CPUs, 4 GB$>$ resources and its unused resources are yielded to A so that A has 24 tasks launched. Next at time t_2, the total number of pending tasks for B is 24, larger than 20. However, due to *memoryless* of DRF, it can only launch 20 tasks. The scenario is similar at t_3, t_4. It is *unfair* for B since the total numbers of tasks scheduled for A and B finally become $86(=24+20+22+20)$ and $56(=4+20+12+20)$ at time t_4, respectively. If this case often occurs, it will be *unfair* for B to consume the amount of resources that she should receive concerning her resource contribution from a long-term perspective.

In contrast, as shown in Table 1b, if we adopt the long-term multi-resource fairness (e.g., H-MRF) as we will introduce in Section 5, the total allocations for A and B are finally the same (e.g., 59), being fair for A and B at t_4 from a long-term point of view.

In summary, DRF cannot satisfy truthfulness and resource-as-you-contributed fairness.

3.3 Asset Fairness (AF)

Besides DRF, there is another popular multi-resource fair policy called *Asset Fairness* [20]. It achieves the fairness by equalizing the aggregation results of all typed resources

TABLE 1
A Comparison Example of *MemoryLess Multi-Resource Fairness* and *Long-Term Multi-Resource Fairness* in a Computing System, Consisting of <100 CPUs, 100 GB$>$ Resources for Two Users A and B

	User A : <1 CPU, 4 GB$>$				User B : <4 CPUs, 1 GB$>$			
	# of Tasks		Allocation		# of Tasks		Allocation	
	New	Total	Running	Total	New	Total	Running	Total
t_1	30	30	24	24	4	4	4	4
t_2	20	26	20	44	24	24	20	24
t_3	27	33	22	66	8	12	12	36
t_4	10	21	20	**86**	30	30	20	**56**

(a) Allocation results based on *Memoryless Multi-resource Fairness* (e.g., DRF, AF). *Total Tasks* refers to the sum of the new arriving tasks and accumulated remaining tasks in previous time.

	User A : <1 CPU, 4 GB$>$				User B : <4 CPUs, 1 GB$>$			
	# of Tasks		Allocation		# of Tasks		Allocation	
	New	Total	Running	Total	New	Total	Running	Total
t_1	30	30	24	24	4	4	4	4
t_2	20	26	4	28	24	24	24	28
t_3	27	49	23	51	8	8	8	36
t_4	10	36	8	**59**	30	30	23	**59**

(b) Allocation results based on *Long-Term Multi-resource Fairness* (e.g., H-MRF).

allocated to each user, with the same assumption as DRF that equal shares of different typed resources are worth the same value. Each time, it performs the fair allocation with the max-min policy [9] by choosing the user with the minimum aggregate resources.

Let us explain AF with *Example 1*. Since the number of CPUs is equal to the GB units of memory (i.e., 100 CPUs and 100 GB memory), we can assume that one GB memory is worth one CPU. If one CPU and one GB memory are both worth \$1, User A costs \$3 for each task, whereas User B costs \$2 for each task. The asset fairness first allocates User A with 28 tasks (i.e., <28 CPUs, 56 GB$>$) and B with 42 tasks (i.e., <42 CPUs, 42 GB $>$) so that their aggregate resources are equal. Next, since there are still <30 CPUs, 2 GB$>$ resources remained and it further allocates 2 tasks for User B to exhaust all remaining memory resources. The final allocation can therefore be 28 tasks (i.e., <28 CPUs, 56 GB$>$) for A and 44 tasks (i.e., <44 CPUs, 44 GB$>$) for B.

Problems of Asset Fairness. While AF seems compelling in its simplicity, there are several significant drawbacks below.

Sharing Incentive Problem. Consider the aforementioned *Example 1*. For User B, there will be 50 tasks allocated in the non-sharing partition of <50 CPUs, 50 GB$>$, better than 44 tasks in the sharing case. Therefore, AF violates sharing incentive property.

Resource-as-you-Contributed Unfairness Problem. Consider again the two-user case of Example 3. Each task of A and B takes \$5 under AF policy. The fairness can be achieved when each of A and B submits 20 tasks. At time t_1, B runs 4 tasks and the remaining resources yields to A such that it has 24 tasks scheduled. After that, since AF is *memoryless*, the allocation results at time t_2 are 20 tasks for each of A and B even though there are 24 tasks demand for B. The case is similar to time t_3 and t_4 and the resulting allocation is shown in Table 1, which illustrates that AF fails to satisfy resource-as-you-contributed fairness.

4 Fairness Definition

In practice, resource sharing can make some users in the shared environment get *sharing benefit* (i.e, more efficiently used resources) over non-sharing. On the other hand, due to the resource contention, resource sharing can also possibly make the total resource a user used smaller than that under non-sharing (i.e., *sharing loss*). For a good sharing policy to achieve the resource-as-you-contributed fairness and sharing incentive property, it should be able to minimize *sharing loss* first and then maximize *sharing benefit* for users.

We define some terminology for a multi-resource allocation system. Suppose the number of resource types is m and the number of users is n. Let $\mathbf{R} = < r_1, \ldots, r_m >$ be the *resource capacity vector* of the system. Let $\mathbf{U}_i(t) = < u_{i,1}(t), \ldots, u_{i,m}(t) >$ be the current *resource allocation vector* at time t, where $u_{i,j}(t)$ is the current volume of resource type j allocated to user $i (1 \le i \le n)$ at time t. Let $\mathbf{D}_i(t) = < d_{i,1}(t), \ldots, d_{i,m}(t) >$ denote the current *resource demand vector* at time t, where $d_{i,j}(t)$ is the current demand of resource type j for user i at time t. Then it holds that $u_{i,j}(t) \le d_{i,j}(t)$, for any resource type j. Let $\mathbf{S}_i(t) = < s_{i,1}(t), \ldots, s_{i,m}(t) >$ represent the *resource fair share vector* at time t, where $s_{i,j}(t)$ is the current fair share of resource type j for user i. Assume the weight for user i is w_i, where $w_i = K_i$. Then user i's resource share ratio is $w_i / \sum_{k=1}^{n} w_k$. Thereby, for user i, her resource share $\mathbf{S}_i(t)$ under the system resource capacity $\mathbf{R}$ can be computed as

$$\mathbf{S}_i(t) = \mathbf{R} \cdot w_i / \sum_{k=1}^{n} w_k. \tag{1}$$

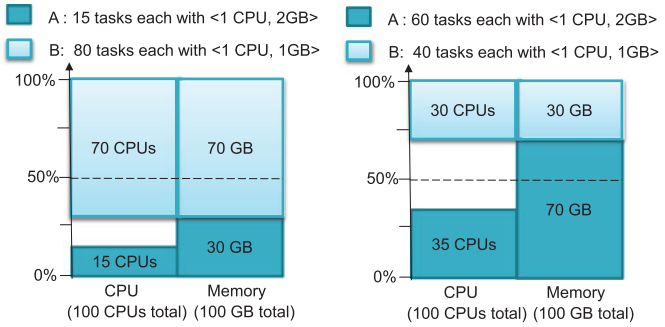
Note that the above runtime information (e.g., $\mathbf{R}, \mathbf{U}_i(t)$, $\mathbf{D}_i(t)$, and $\mathbf{S}_i(t)$) can be obtained during the computation in current reservation-based systems such as YARN.

In the shared environment, due to the resource contention between users, $u_{i,j}(t)$ can be larger or smaller than $s_{i,j}(t)$. In contrast, in the non-shared environment, since there is no resource contention issue, it thereby holds that $u_{i,j}(t) = \min\{d_{i,j}(t), s_{i,j}(t)\}$ for any resource type j. For each resource type $j (1 \le j \le m)$, we define the sharing (fairness) degree $\beta_{i,j}(t)$ for the ith user as follows:

$$\begin{aligned} \beta_{i,j}(t) &= \frac{AllocationWithSharing}{AllocationWithoutSharing} \\ &= \frac{\int_0^t u_{i,j}(t)\, \mathbf{d}t}{\int_0^t \min\{d_{i,j}(t), s_{i,j}(t)\}\, \mathbf{d}t}. \end{aligned} \tag{2}$$

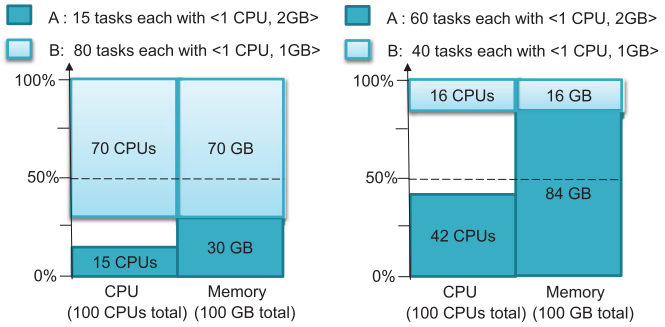
Moreover, in multi-resource allocation, since resources of different types (e.g., CPU and memory) cannot be converted between each other, the number of tasks that can be allocated is determined by the dominant (bottleneck) resource type [20]. Thus, to have more tasks allocated in resource sharing than non-sharing (i.e., sharing benefit), it must hold that $\beta_{i,j}(t) \ge 1$ for any resource type j. Formally, we define the sharing (fairness) degree $\beta_i(t)$ for the ith user at time t as follows:

$$\beta_i(t) = \min_{1 \le j \le m} \beta_{i,j}(t). \tag{3}$$



(a) Allocation at t_1. (b) Allocation at t_2.

Fig. 3. An example showing LT-DRF resource allocation.



(a) Allocation at t_1. (b) Allocation at t_2.

Fig. 4. An example showing LT-AF resource allocation.

For user i in the shared environment, $\beta_i(t) > 1$ indicates the sharing benefit for the ith user at time t. $\beta_i(t) < 1$ means the sharing loss. In contrast, it always holds $\beta_i(t) = 1$ in the non-shared environment, since it has $u_{i,j}(t) = \min\{d_{i,j}(t), s_{i,j}(t)\}$ at any time t. That is, there are neither sharing benefit nor sharing loss for users in the non-shared environment. Therefore, for a good fair policy, it should be able to achieve $\beta_i(t) \geq 1$ for all users (i.e., sharing benefit) in the shared environment from a long-term view.

5 Long-Term Multi-Resource Fairness

In Sections 3.2 and 3.3, we have shown that both DRF and AF are memoryless, which perform the fairness allocation without considering historical allocation. Because of this, they fail to satisfy the resource-as-you-contributed fairness and are thus not suitable for pay-as-you-use computing systems. To address it, in the following sections, we attempt to investigate the *time* dimension of multi-resource allocation by extending the existing multi-resource policies (e.g., DRF and AF) with the long-term notion. Unfortunately, both attempts are still unable to satisfy all the desired properties of resource sharing in the pay-as-you-use computing system in Section 2. Finally, we propose a new algorithm named H-MRF to achieve the fairness with provable properties.

5.1 Attempt 1: Long-Term Dominant Resource Fairness (LT-DRF)

Our first try is to enhance the existing (*memoryless*) multi-resource allocation policy DRF with the long-term notion. We name the resulting *spatial-temporal* fairness policy as *Long-Term Dominant Resource Fairness*. It focuses on the accumulated multiple resources, defined as the sum of currently allocated multiple resources and historical allocated multiple resources. For each user, LT-DRF calculates the share of each accumulated resource consumed by that user. The maximum among all shares of a user is called that user's *accumulated dominant share*, and the corresponding accumulated resource is referred to as the *accumulated dominant resource*. LT-DRF seeks to maximize the smallest accumulated dominant share and achieves the multi-resource fairness by equalizing the accumulated dominant shares across all users.

Example 4. We extend Example 1 by assuming that initially at time t_1, the workloads for A are 15 tasks and for B are 80 tasks. At time t_2, there are 60 tasks for A and 40 tasks for B to run.

Consider the two-user case of Example 4. Fig. 3 illustrates the allocation results with LT-DRF. At time t_1, A has a small number of tasks and thus her unused resources are released to B, as shown in Fig. 3a. The accumulated dominant shares for A and B are 30 and 70, respectively. In contrast, at time t_2, LT-DRF allocates more resources to A (i.e., schedules 35 tasks for A and 30 tasks for B) so as to make the accumulated dominant shares for A and B become the same of 100, as shown in Fig. 3b.

Theorem 1. *LT-DRF is truthfulness.*

Let's revisit the former Example 2. In contrast to DRF policy under which cheating users can get benefits as shown in Fig. 2b, C's cheating for preempting resources under LT-DRF policy is a pre-consumption of its own resources and needs to *pay back* at a later time to others, which does benefit itself at all in the long run.

Theorem 2. *LT-DRF violates the resource-as-you-contributed fairness property.*

The detailed proofs for Theorems 1 and 2 are presented in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2017.2788880. In summary, although LT-DRF addresses the untruthfulness problem of DRF and inherits the sharing incentive property of DRF, it still fails to satisfy the resource-as-you-contributed fairness as given by Theorem 2.

5.2 Attempt 2: Long-Term Asset Fairness(LT-AF)

Another attempt is to extend Asset Fairness with the long-term notion. We call the resulting policy *Long-Term Asset Fairness*.

The idea of LT-AF is to equalize the aggregate resource share of accumulated multiple resources allocated to each user. Particularly, for each user i, LT-AF computes the aggregate accumulated share $\alpha_i = \sum_{1=j}^{m} \{\ddot{u}_{i,j}/r_j\}$, where $\ddot{u}_{i,j}$ is the share of accumulated resource j given to user i. It then allocates resources with max-min by repeatedly choosing the user with the minimum aggregate accumulated share.

Let's consider again the same Example 4 of Section 5.1. Fig. 4 illustrates the resource allocation result with LT-AF. At time t_1, LT-AF launches 15 tasks for A and 70 tasks for B by releasing unused resources from A to B. Then the aggregate accumulated shares for A and B are $45(= 15 \times 1 + 15 \times 2)$ and $140(= 70 \times 1 + 70 \times 1)$, respectively. However, at t_2, since B's accumulated share at t_1 is larger than A, LT-AF schedules

42 tasks for A and 16 tasks for B, so that aggregate accumulated shares are close to each other between A (i.e., 171) and B (i.e., 172).

Theorem 3. *LT-AF satisfies resource-as-you-contributed fairness property.*

Theorem 4. *LT-AF violates sharing incentive property.*

For the proofs of Theorems 3 and 4, please refer to Appendix B, available in the online supplemental material. To summarize, although LT-AF overcomes the resource-as-you-contributed unfairness problem of AF, it still violates the sharing incentive property.

5.3 Our Solution: Hybrid Multi-Resource Fairness (H-MRF)

Addressing the limitations of the previous two attempts, we derive a new algorithm for achieving the multi-resource fairness, which we call H-MRF.

5.3.1 H-MRF Algorithm

Attempt 1 and Attempt 2 indicate that the naive extensions of existing *memoryless* multi-resource fairness policies (e.g., DRF, AF) with the long-term notion have serious fairness problems in the pay-as-you-use computing system. However, we make an interesting observation that they are complementary to each other. For Attempt 1, LT-DRF extension inherits the sharing incentive property of DRF, but fails to satisfy the resource-as-you-contributed fairness. In contrast, LT-AF in Attempt 2 satisfies the resource-as-you-contributed fairness, whereas it violates the sharing incentive property. It might be possible for us to have a new policy by combining DRF and AF with the long-term notion such that it can satisfy all the desired properties for resource sharing in the pay-as-you-use computing system.

The basic idea of H-MRF is that, if we can have a mechanism to detect when LT-AF violates sharing incentive (i.e., there are some sharing-loss users) dynamically at runtime, we can allocate more resources to those sharing-loss users so that from a long-term view, all users are not sharing loss. Then the modified policy will satisfy sharing incentive and resource-as-you-contributed fairness.

Based on this idea, we propose *H-MRF*, a novel combination of DRF and AF with the long-term notion. The pseudocode is presented in Algorithm 1. Each time, H-MRF performs the resource allocation through the following four steps:

Step 1#. Computes the sharing degree β_i for each user according to Formula (3) in Section 4. If there are sharing-loss users (i.e., $\beta_i < 1$), it allocates resources to the user with the minimum β_i for sharing incentive purpose allocation (Lines 10-11). Otherwise, it indicates that all users get sharing benefits and thus moves to Step 2 for resource-as-you-contributed fairness purpose allocation (Lines 12-16).

Step 2#. Computes the aggregation value of total resources allocated to every user over time in order for finding a set of users N with the minimum aggregation value of accumulated resources (Lines 13-14).

Step 3#. Finds the user with the minimum dominant resource over the user set N (Line 15).

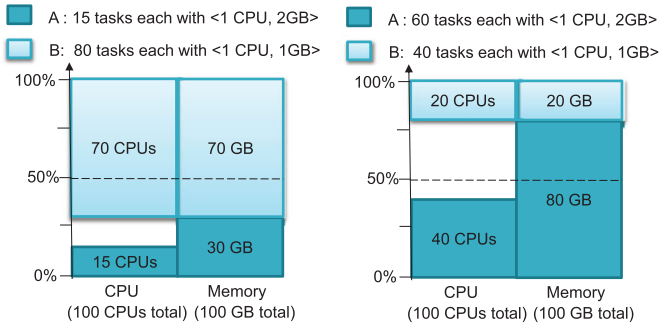
Step 4#. Allocates resources to the user from N (Line 16).

Algorithm 1. Hybrid Multi-Resource Fairness of DRF and AF

1: $\mathbf{R} = <\mathbf{r_1}, \ldots, \mathbf{r_m}>$**:** total resource capacities.
2: $\mathbf{C} = <\mathbf{c_1}, \ldots, \mathbf{c_m}>$**:** consumed resources, initially 0.
3: $\mathbf{U_i} = <\mathbf{u_{i,1}}, \ldots, \mathbf{u_{i,m}}>$ $(\mathbf{i} = \mathbf{1} \ldots \mathbf{n})$**:** currently allocated resources for user i.
4: $\ddot{\mathbf{U}}_i = <\ddot{u}_{i,1}, \ldots, \ddot{u}_{i,m}>$ $(i = 1 \ldots n)$**:** accumulated resources consumed by user i.
5: $\mathbf{S_i} = <\mathbf{s_{i,1}}, \ldots, \mathbf{s_{i,m}}>$ $(\mathbf{i} = \mathbf{1} \ldots \mathbf{n})$**:** current resource share for user i.
6: $W = <w_1, \ldots, w_n>$: weighted share. w_i denotes the weight for user i.
7: β_i: the sharing degree for user i (See Formula (3) in Section 4).
8: **while** there are pending tasks **do**
9: **Find** user i satisfying $\beta_i = \min_{1 \le j \le n} \beta_j$.
10: **if** $\beta_i < 1$ **then** ▷ Sharing loss for user i
11: *ALLOC(i).* ▷ Allocate resources to user i.
12: **else** ▷ All users have sharing benefits.
13: **Get** a set N of users satisfying the following condition:
14: $N = \{i | \sum_{j=1}^{m} \ddot{u}_{i,j} \le \min_{1 \le k \le n} \sum_{j=1}^{m} \ddot{u}_{k,j}\}$. ▷ Users with minimum overconsumption of resources. (AF policy)
15: **Choose** user $i (i \in N)$ with the smallest weighted dominant share $max_{j=1}^{m}\{\ddot{u}_{i,j}/(r_j \cdot w_i)\}$. ▷ DRF Policy.
16: *ALLOC(i).* ▷ Hybrid resource allocation with AF and DRF.
17: **function** ALLOC(User i)
18: $\mathbf{D}_i \leftarrow$ the next task demand for user i. ▷ $\mathbf{D}_i = <d_{i,1}, \ldots, d_{i,m}>$.
19: **if** $\mathbf{D}_i + \mathbf{C} \le \mathbf{R}$ **then**
20: $\mathbf{C} = \mathbf{C} + \mathbf{D}_i$. ▷ Update currently consumed resources.
21: $\mathbf{U}_i = \mathbf{U}_i + \mathbf{D}_i$. ▷ Update user i's currently consumed resources.
22: **for each** k in $\{1 \ldots m\}$ **do** ▷ Update accumulated resources.
23: $\ddot{u}_{i,k} = \ddot{u}_{i,k} + \min\{u_{i,k}, s_{i,k}\} + \max\{0, u_{i,k} - s_{i,k}\}$.
24: update β_i based on Formula (3) in Section 4.
25: Allocate resource to user i.
26: **else** ▷ The system is utilized at maximum.
27: **Wait** until there is a completed task of resources $\mathbf{R}_i$ from user i.
28: $\mathbf{C} = \mathbf{C} - \mathbf{R}_i$. ▷ Update current system resource usage.

In the above resource allocation, Step 1 tries to make sure that in the long-term resource allocation all users can get sharing benefit over the exclusively non-sharing case, which is a requirement for sharing incentive purpose and it is given the first priority. Only if that is guaranteed, we turn to Step 2, 3 and 4 for resource-as-you-contributed fairness purpose allocation. Step 2 seeks for a set of users with the minimum aggregation value of accumulated resources and Step 3 picks up the user with the minimum dominant resource over such a user set N. Finally, Step 4 performs the resource allocation to the selected user from Step 3.

Consider the two-user case of Example 4 in Section 5.1. Fig. 5 shows the resource allocation results with H-MRF. At t_1, H-MRF launches all 15 tasks for A and 70 tasks for B. The total aggregate resource value for A is 45, and for B is 140. The sharing degrees for A and B are $\beta_A(t_1) = 1, \beta_B(t_1) = 1.4$, respectively. However, at time t_2, both of them have large resource demands. To achieve resource-as-you-contributed



(a) Allocation at t_1. (b) Allocation at t_2.

Fig. 5. An example showing H-MRF resource allocation. At t_1, H-MRF schedules 15 tasks for A and 70 tasks for B, and the sharing degrees are $\beta_A(t_1) = 1, \beta_B(t_1) = 1.4$. The total aggregate resource value for A and B are 45, 140, respectively (i.e., the difference is 95). At t_2, H-MRF schedules 40 tasks for A and 20 tasks for B so that the difference of total aggregate resource values between A and B are minimized to 15 subject to $\beta_A(t_2) \geq 1 \wedge \beta_B(t_2) \geq 1$ (i.e., $\beta_A(t_2) = 1.375, \beta_B(t_2) = 1$).

fairness, H-MRF allocates more resources to A so that their total aggregate resource values are close to each other (i.e., 165 for A, 180 for B) in the case that all users are not sharing loss. The sharing degrees at time t_2 are $\beta_A(t_2) = 1.375$ and $\beta_B(t_2) = 1$ for A and B, respectively.

Finally, we show that H-MRF satisfies all desired properties list in Section 2.

Theorem 5. *H-MRF satisfies sharing incentive property and resource-as-you-contributed fairness.*

Theorem 6 (Truthfulness). *A user cannot have more amount of resources/tasks allocated in H-MRF by falsely reporting her true demand.*

Theorem 7 (Pareto efficiency). *A user cannot increase her resources/tasks allocation in H-MRF without decreasing other users' allocation when system resources are fully utilized.*

The detailed proofs of the aforementioned Theorems 5, 6 and 7 are given in Appendix C, available in the online supplemental material.

Moreover, we also extend H-MRF for the distributed scenario and show that it meets all desired properties listed in Section 2. The detailed extension and the proof for each property of H-MRF in distributed resource allocation can be found in Appendix D, available in the online supplemental material.

Finally, we summarize the fairness properties that are satisfied by DRF, AF, LT-DRF, LT-AF and H-MRF in Table 2. Only H-MRF can satisfy all the desired properties in the pay-as-you-use computing system.

TABLE 2
Properties of DRF, AF, LT-DRF, LT-AF and H-MRF

Property	Allocation Policy				
	DRF	AF	LT-DRF	LT-AF	H-MRF
Sharing Incentive	√		√		√
Resource-as-you-contributed Fairness				√	√
Truthfulness			√	√	√
Pareto Efficiency	√	√	√	√	√

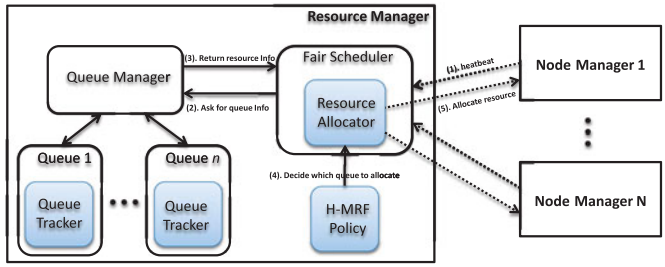


Fig. 6. Overall design of MRYARN. New components are shown in rectangle with blue background, and others are from YARN.

6 Implementation on YARN

YARN [4] has been a popular resource management system that enables a number of data-intensive computing frameworks (e.g., MapReduce [16], Spark [40], HIVE [35]) to efficiently share a cluster. In this section, we implement H-MRF policy in YARN. Particularly, we develop a prototype called *MRYARN* (Multi-Resource YARN) in about 1,500 lines of code. Fig. 6 illustrates the overall design of MRYARN. We add two major new components on top of YARN Resource Manager, namely, *Queue Tracker (QT)* and *Resource Allocator (RA)*. QT tracks and monitors allocation information of each queue over time. Based on the provided runtime information, RA decides and allocates multiple resources to queues dynamically.

Queue Tracker. To enable resource sharing, YARN organizes resources into multiple queues. Each queue can represent a user or an organization. For MRYARN, to achieve the dynamic multi-resource allocation across multiple queues, there is a need to maintain the runtime information for each queue. We add a Queue Tracker inside each queue to achieve that. Particularly, according to H-MRF, two key information should be kept for each queue, namely, *accumulated resources* and *fairness degree*.

Time Window-based Support. Instead of keeping the long-term multi-resource fairness all the time since the system starts, our MRYARN supports the long-term fairness over a period of time (e.g., 1 day) for the sake of different users' needs. Typically, two types of time window are supported as follows:

(I) *Tumbling Window*. It divides the whole time into a set of disjoint time windows of length L_t and ensures 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 resource fair allocation from the beginning (Inter-window allocation). That is, the tumbling window-based MRYARN is a hybrid scheduler with the *memorizing* scheduling within each time window, and *memoryless* scheduling across time windows.

(II) *Sliding Window*. Unlike tumbling window, it consists of a number of (overlapped) time windows. Each time window is created when a task scheduling event is triggered. Given a window length L_s and the current scheduling time t_c, it only maintains the resource allocation within the current time window $[t_c - L_s, t_c]$. For historical allocation information, only those jobs whose completion time are within the current time window are counted. It performs

the long-term multi-resource fairness within the current time window (Intra-window allocation), and drops the historical allocation information for memoryless allocation from previous time windows (Inter-window allocation).

For the tumbling window-based (or the sliding window-based) MRYARN, when $L_t \to +\infty$ (or $L_s \to +\infty$), it turns out to be a pure *long-term* fair scheduler that memorizes all the historical resource allocations since YARN system starts. In contrast, it becomes a *memoryless* fair scheduler for MRYARN when $L_t \to 0$ (or $L_s \to 0$). It indicates that MRYARN is also a generalization of pure *long-term* and *memoryless* schedulers.

Resource Allocator. Resource Allocator is responsible for performing the multi-resource allocation to each queue. The H-MRF policy is implemented in RA. When there are idle resources and pending tasks, RA determines how much resources should be given to which queue dynamically, based on H-MRF algorithm and runtime allocation information provided by QT. Moreover, to make it flexible, we provide users with a key argument of time window in the default configuration file, to allow users to set it in terms of their requirements.

Efficient Task Placement. In multi-resource allocation, resource fragmentation can easily occur for a machine without proper task placement [22], which results in poor resource utilization and performance. Given a task and multiple machines with sufficient idle resources, RA needs to carefully choose a machine for the task so as to minimize the overall fragmentation of the cluster. Grandl et al. [22] have shown that it is analogous to multi-resource bin packing, which is however a *NP*-problem. We thus propose a heuristic approach by defining the *affinity* of a task as follows: Given a set of machines for a task, we first compute an *affinity value* for each machine to the task. The affinity value is the dot product between the task's resource demand and the vector of the machine's idle resources. The machine with the highest affinity score is chosen for the task.

Overall Resource Allocation Flow. When a node manager having idle resources connects to the resource manager in a heartbeat, the fair scheduler will ask queue manager for the resource allocation information of each queue, provided by QT located at each queue. Having the resource allocation information, RA then determines which queue to allocate based on its H-MRF policy, and performs the allocation finally.

6.1 Practical Considerations for MRYARN

Although the long-term fair policy H-MRF has many merits, there are still some practical problems that need to be addressed for MRYARN.

Starvation and Resource Oscillation Problems. As a long-term fair policy, H-MRF allocates resources fairly based on the historical resource allocations for users. It is thereby prone to occur in MRYARN that a user yields her resources for a long time (e.g., 30 mins) and then grabs lots of resources, starving other users for a long while without being allocated (i.e., *Starvation Problem*). Moreover, it is also possible that an adversarial user can 'oscillate' the resource allocation of the system deliberately under the long-term fair scheduling (i.e., *Resource Oscillation Problem*). For example, in the long-term fair scheduling, the adversarial user can pile-up resources and then forces other users out (during hot times).

To address these two problems, we propose a starvation-aware and resistant scheduler for MRYARN. It is based on the *time-out* technique. The core idea is that, we provide users with a threshold $t_{timeout}$ and allow that any queue (user) who has been waiting for $t_{timeout}$ since its last time allocation is given the highest priority in resource allocation. Through time-out checking and dynamic resource allocation, we can alleviate the starvation problem and meanwhile make scheduler resistant to adversarial behaviors from users. By default, $t_{timeout}$ is initialized to the length of time window. Users can configure it according to their needs.

Time Window Issue. As we have mentioned earlier, the window-based MRYARN is a generalized scheduler with memorizing scheduling within the time window and memoryless scheduling across time windows. It means that the historical resource allocation of previous windows will be no longer counted in the current window. Because of this, it will offer hostile users ample opportunity to unfairly use resources. For example, given a 1-hour window under the tumbling window-based MRYARN, the hostile users have to do is to run tasks preempting (borrowing) resources for just a few minutes during each window, which will allow them to earn lots of resources, especially when there are too many time windows.

To alleviate this problem, one practical solution is to enlarge the time window length so that there are a fewer number of time windows. For example, if the shared YARN cluster will run 24 hours, we can change the time window length from 1 hour to a bigger value (e.g., 4 hours). Another possible solution is to have a limitation on the maximum of preempted (borrowed) resources for a user in each time window.

7 Experimental Evaluation

We have used two complementary methods to evaluate the effectiveness of our proposed approach. We first evaluate H-MRF using our prototype MRYARN on an Amazon EC2 cluster. To estimate H-MRF at larger scale, we further conduct trace-driven simulations using Google cluster-usage traces, as the results given in Appendix F, available in the online supplemental material.

7.1 Experiment Setup

YARN Cluster. We have implemented H-MRF in the version of YARN-2.4.0. We deploy the YARN framework in an Amazon EC2 cluster consisting of 60 Amazon EC2 m3. xlarge instances each with 4 virtual cores and 15 GB memory. We configure 1 instance as master, and the remaining 59 instances as slaves, each of which is configured with < 4 virtual cores, 15 GB $>$.

Macro-Benchmark. We run a mix of four different workloads for MRYARN below. Particularly, Facebook, Purdue, and TPC-H are *data-intensive* workloads, whereas Spark workload is *compute-intensive* one.

- *Synthetic Facebook Workload:* We synthesize Facebook workload based on the distribution of jobs sizes and inter-arrival time at Facebook provided by Zaharia et al. [39]. 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

TABLE 3
Job Types and Sizes for Synthetic Facebook Workloads

Bin	Job Type	Map Tasks		Reduce Tasks		# Jobs
		#	Demand	#	Demand	
1	rankings selection	1	<1,1 GB>	NA	NA	38
2	grep search	2	<1, 1.5 GB>	NA	NA	18
3	uservisits aggregation	10	<2, 0.5 GB>	2	<4,2 GB>	14
4	rankings selection	50	<4, 1 GB>	NA	NA	10
5	uservisits aggregation	100	<2, 1.5 GB>	10	<2, 2 GB>	6
6	rankings selection	200	<3, 2 GB>	NA	NA	6
7	grep search	400	<2, 1 GB>	NA	NA	4
8	rankings-uservisits join	400	<1, 2 GB>	30	<2, 0.5 GB>	2
9	grep search	800	<2, 0.5 GB>	60	<1, 3 GB>	2

number of small-sized jobs ($1 \sim 15$ 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) [7]. The demand distribution of map/reduce tasks is based on Fig. 1 provided by Ghodsi et al. [20]. The jobs are from Hive benchmark [2], containing four types of applications, i.e., rankings selection, grep search (selection), uservisits aggregation and rankings-uservisits join.

- *Purdue Workload:* Five benchmarks (e.g., WordCount, TeraSort, Grep, InvertedIndex, HistogramMovices) are randomly chosen from Purdue MapReduce Benchmarks Suite [13]. We use 40G wikipedia data [11] for WordCount, InvertedIndex and Grep, 40G generated data for TeraSort and HistogramMovices with provided tools. To emulate a series of regular job submissions in a data warehouse, we submit these jobs sequentially at an interval of 3 mins to the system.
- *TPC-H Workload:* To emulate continuous analytic query, such as analysis of users' behavior logs, we ran TPC-H benchmark queries on Hive [3]. 40 GB 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 Workload:* We choose three compute-intensive machine learning algorithms, namely, kmeans, linearRegression(LR), 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 1) each kmeans instance with 100 workers, each with <2 CPUs, 2 GB> resources; 2) each LR instance with 150 workers with <2 CPU, 1 GB> resources per worker; 3) each ALS instance with 100 workers, each with <1 CPU, 2 GB>.

We assume that there are four users User 1, User 2, User 3 and User 4, running Facebook, Purdue, Spark and TPC-H workloads on the YARN cluster with equal resource contributions, i.e, each user pays for 15 computing nodes of the YARN cluster. We use the macro-benchmark to evaluate the resource sharing fairness and performance for H-MRF in Sections 7.2.2 and 7.2.3.

Micro-Benchmark. We create two queues namely, *Queue1* and *Queue2* with equal share. We run two jobs from Purdue benchmarks each with the input wikipedia data of 40 GB. Job 1 is Sort (i.e., memory-intensive) and Job 2 is WordCount (i.e., CPU-intensive). Each of them have 640 map tasks and 200 reduce tasks. YARN is container based. We configure Job 1 with <2 CPUs, 1 GB> per map task and <3 CPUs, 4 GB> per reduce task. In contrast, Job 2 is configured with <4 CPUs, 1 GB> per map task and <2 CPUs, 4 GB> per reduce task. We submit Job 1 first to Queue1 and wait for 60 seconds before submitting Job 2 to Queue2. We use this micro-benchmark to show the dynamic resource sharing process for H-MRF in Section 7.2.1.

Trace-Driven Simulator. To evaluate H-MRF at a larger scale, we developed a trace-driven simulator that replays logs from Google clusters. The simulator mimics these aspects of tasks from the original trace: task submission time, task resource requirements (e.g., cpu, memory) and execution time, which are the least required information needed for any task scheduling simulator.

Trace Dataset. Originally, the Google traces provide the information about tasks submitted by over 900 users on a cluster of $12K$ machines in one month, which are specified by *job_events*, *task_events*, *machine_events*, *machine_attributes*, *task_constraints*, *task_usage* listed in the schema.csv file. The data of our simulator are retrieved from *task_events (user, task_index, cpu_request, memory_request,...)* and *task_usage (user, task_index, start_time, end_time)* tables, both of which contain a common attribute of *user* and *task_index*. To generate our dataset, we first sort *task_events* according to *user* attribute. Next, we select tasks of the first one hundred users from *tasks_events* and find the corresponding task *start_time* and *end_time* from *task_usage* according to *task_index*. The *execution time* is calculated through *end_time* minus *start_time*. However, the Google trace does not provide the *task submission time*. In order to make simulator work, we make an assumption by letting *start_time* represent *task submission time*.

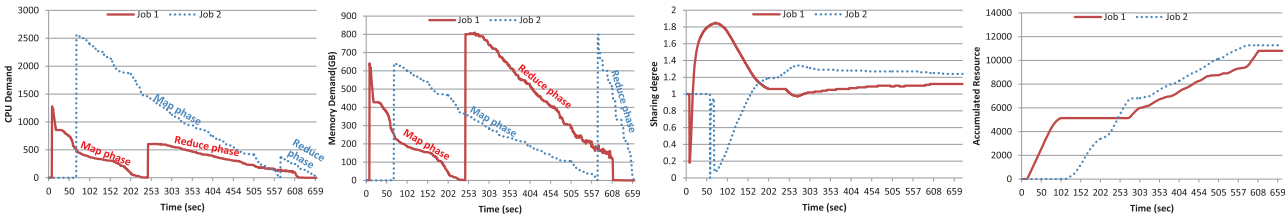
7.2 YARN Benchmark Results

This section evaluates H-MRF in YARN cluster. We start by showing how H-MRF adjusts the resource allocation of jobs with different resource demands dynamically in Section 7.2.1. In Section 7.2.2, we give the resource sharing results for H-MRF. We also compare it with other alternative policies. Finally, we present the performance results and monetary cost of H-MRF as well as alternative polices in Section 7.2.3 and Appendix E, available in the online supplemental material, respectively.

7.2.1 Dynamic Resource Sharing

We first show how H-MRF dynamically allocates resources between jobs and achieves the resource-as-you-contributed fairness by using micro-benchmark. Figs. 7a and 7b show the CPU and memory demand for each MapReduce job during its map/reduce phase computation over time, whereas

3. We reduce the size of the largest jobs in [39] to have the workload fit our cluster size.



(a) CPU demand for Job 1 and Job 2 in map/reduce phase over time.
(b) Memory demand for Job 1 and Job 2 in map/reduce phase over time.
(c) Sharing degree
(d) Aggregate accumulated resource

Fig. 7. The dynamic fair resource allocation process for two jobs under H-MRF, where Job 1 is submitted one minute before Job 2. The graphs (a) and (b) present the CPU and memory demand for each job over time, respectively. The graphs (c) and (d) show the sharing degree and aggregate accumulated resource for each job, respectively.

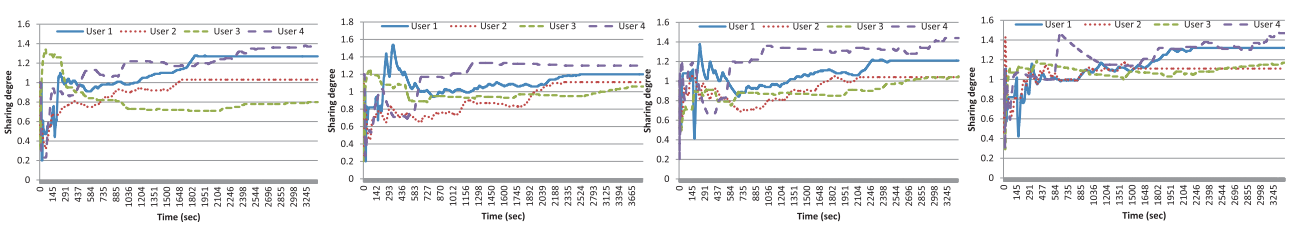
Figs. 7c and 7d present the sharing degree and aggregate accumulated resource. In the first 1 minute, only Job 1 is running on the system and it possesses the whole cluster. As shown in Fig. 7c, the sharing degree goes above 1.0 (i.e., sharing benefit) and achieves the maximum point of 1.84 at the 60th second. After 1 minute, Job 2 is submitted. During the period of $60 \sim 158$ seconds, the sharing degree for Job 2 is below 1. This is because that when Job 2 arrives at time 60th second, the resources have being possessed by Job 1, and then Job 2 needs to wait for idle resources released by the completed tasks from Job 1. During this period, the allocated resources for Job 2 are smaller than its share and demand, making Job 2 have a sudden drop on its sharing degree when it starts running. Observing that Job 2 is under sharing loss, H-MRF gives a higher priority for Job 2 in resource allocation and it makes the sharing degree curve of Job 2 begins to move up smoothly. Note after 158 seconds, both sharing degrees of Job 1 and Job 2 are above 1, despite their dynamic demands. However, H-MRF still gives a higher priority for Job 2, since it detects that the aggregate accumulated resource for Job 2 is smaller than Job1 until the 247th second. During this period, Job 2 possesses most parts of cluster resources by preempting resources from Job 1, which can be observed from the drop of sharing degree curve of Job 1 in Fig. 7c and the flat of aggregate accumulated resource curve for Job 1 in Fig. 7d during the period of $60 \sim 247$ seconds. We observe in Figs. 7a and 7b that there is a sudden high CPU and memory demand at 245th second for Job 1 since its reduce tasks begin to run. Its sharing curve continues to drop during $247 \sim 273$ seconds although its aggregate accumulated resources for Job 1 is smaller than Job 2, since it needs to wait for idle resources released by the completed tasks from Job 2. After that, Job 1 begins to catch up with Job 2. Finally, their aggregate accumulated resources are much close to each other (i.e., Job 1: 10,806, Job 2: 11,270) at the 670th second (i.e., achieving the resource-as-you-contributed fairness).

7.2.2 Resource Sharing Benefit/Loss

We compare DRF, LT-DRF, LT-AF and H-MRF with macro-benchmark in Fig. 8. All results are relative to the non-sharing scenario in which the sharing degree is one. Figs. 8a and 8d show the sharing degree $\beta_i(t)$ for each user over time according to Formula (3). Typically, for the ith user, $\beta_i(t) \geq 1$ indicates sharing benefit. Otherwise, it means sharing loss. We make the following observations:

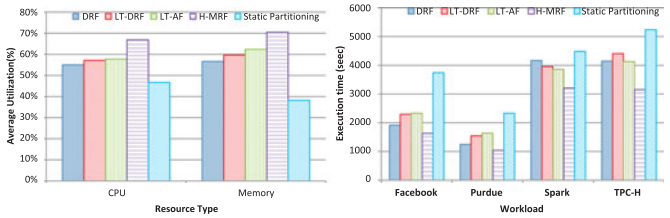
First, resource sharing can benefit most users. Due to the different resource demands for users. Resource sharing can enable overloaded users have chances to possess unused resources from underloaded users, getting more resource allocations than its share at a time and thus better than running in a non-shared partition cluster with maximum allocation of her share at any time.

However, without a proper sharing policy, resource sharing can also possibly downgrade the performance for some users (i.e., the sharing degree is below one) in the pay-as-you-use computing system, which is a very serious problem and its sharing loss should be reduced as much as possible according to SLA requirement. By comparing Figs. 8a, 8b, 8c, and 8d, we can observe that such degradation in DRF is the worst. The sharing loss problem constantly exists until all the users finish. DRF is a *memoryless* fair policy, which does not take into account the historical allocation. Since the demand for each user is changing over time, some users can use much more resources (e.g., User 4 in Fig. 8a) over time whereas some other users (e.g., User 3 in Fig. 8a) under sharing get much fewer resources than that under non-sharing over time. In contrast, LT-DRF, LT-AF and H-MRF are *long-term* fair policies. They adjust the resource allocation for each user dynamically based on the historical



(a) DRF.
(b) LT-DRF.
(c) LT-AF.
(d) H-MRF.

Fig. 8. The comparison of resource sharing results for different policies in YARN. The results are relative to the non-sharing scenario in which the the sharing degree is one. The graphs (a)$\sim$(d) show the detailed sharing degree $\beta_i(t)$ for each user under the corresponding policy according to Formula (3). Typically, for the ith user, $\beta_i(t) \geq 1$ means sharing benefit. $\beta_i(t) < 1$ indicates sharing loss.



(a) Average resource utilization. (b) Execution time.

Fig. 9. The comparison on resource utilization and performance for DRF, LT-DRF, LT-AF, H-MRF and static partitioning (i.e., non-sharing case).

allocation so that the aggregate accumulated resources for each user are close to each other.

Second, H-MRF has a better allocation result (i.e., less sharing loss) than LT-DRF and LT-AF. Different from LT-DRF and LT-AF in resource allocation, H-MRF is sensitive to sharing loss problem. It monitors the sharing degree for each user dynamically and always assigns higher priority to those users under sharing loss (i.e., sharing degree is below one) if available than users under sharing benefit. For example, in Fig. 8d, when H-MRF detects at time 85th second that the sharing degree for User 2 is smaller than one (i.e., sharing loss), it gives a higher priority in resource allocation so that more resources can be allocated for User 2 over time, to maximize the sharing degree (i.e., by reducing the sharing loss). Note, there are sharing loss for some users (e.g., User 1) at the beginning of computation in Fig. 8d. This is due to the unavoidable resource waiting problem (for any policies): the tasks of the first arriving user possesses the whole cluster resources, which makes the tasks of later arriving users have to wait for idle resources released by former user's tasks for a small amount of time.

7.2.3 Performance Evaluation

Fig. 9 presents the average resource utilization and the performance for each workload of the macro-benchmark. We implement the static partitioning in YARN for four workloads by dividing the whole cluster resources equally into four portions and limiting the maximum size of each portion (i.e., memory and CPU resources) that can be allocated to each queue. We have the following observations:

First, resource sharing is better than static partitioning in resource utilization and performance. As shown in Fig. 9a the average CPU and memory utilizations for sharing policies (e.g., DRF, LT-DRF, LT-AF, H-MRF) are higher than that in static partitioning. For example, the average memory utilizations for DRF, LT-DRF, LT-AF, H-MRF are $57\%, 59\%, 62\%, 71\%$, respectively, whereas there is only 38 percent for static partitioning. As shown in Fig. 9b, each workload consumes longer time to finish in the static partitioning than in the sharing case. For example, for Facebook workload, H-MRF is $2.1\times$ faster than static partitioning. The performance gain is primarily due to the resource preemption of unused resources from overloaded queues in the sharing case, which can be reflected from the higher CPU/memory resource utilization in the sharing. The observation is consistent with prior works such as [23].

Second, H-MRF outperforms other baseline sharing policies in performance due to its efficient task placement in reducing the fragementation of machines in multi-resource allocation, whereas other alternative sharing policies do not have such a concern and simply treat all machines as a single super machine.

8 Related Work

Multi-Resource Fairness. As for multi-resource allocation, DRF is a popular fair policy [20], which provides fair allocation of multiple resources based on dominant shares. The attractiveness of DRF stems from its good merits such as sharing incentive, envy-freeness, and pareto-efficiency. It has been implemented in many current datacenter framework, including YARN [4], Mesos [23]. After that, there have been lots of extension and generalization for DRF. Bhattacharya et al. [14] generalized DRF to support hierarchical scheduling. Ghodsi et al. [19] extended DRF to fair queueing of package processing. Wong et al. [25] later generalized the measure of DRF and incorporated it into a unifying framework that captures the tradeoffs between performance efficiency and fairness. Kash et al. [26] extended the DRF model to a dynamic setting where users can join the system over time but will never leave. Wang et al. [36] generalized DRF in a distributed system with heterogeneous servers, followed by a TSF fairness policy for the case when there is a placement constraint for task placement [37]. Dolev et al. [18] generalized DRF to consider multiple contended resources by proposing bottleneck-based fairness, rather than the single dominant (bottleneck) resource only for each user. Parkes et al. [28] extended DRF in several ways and focused on in particular the case of indivisible tasks. Liu et al. [27] proposed a Reciprocal Resource Fairness by extending DRF to allow the trade among different types of resources between users. Tang et al. [30] proposed a EMRF to balance the tradeoff between fairness and efficiency for Coupled CPU-GPU architecture by extending DRF with the knob. When the resource demand vector required by DRF is not available in computer architectures, Zahedi et al. [42] proposed an alternative multi-resource policy based on Cobb-Douglas utility function for multiprocessors. However, all the works above are memoryless, i.e., allocating resource at the instant time without historical allocation information considered. We show particularly in Section 3.2 that it encounters severe problems in the pay-as-you-use computing system, due to its *memoryless* property. In comparison, we proposed a *spatial-temporal* multi-resource fairness policy called H-MRF and showed that it can address all these problems and meanwhile satisfy all the properties desired for pay-as-you-use computing.

Fair Schedulers in Data Processing Systems. Many data processing systems now support the fair scheduling of multiple jobs, such as Hadoop Fair Scheduler [39], Quincy [24], Mesos [23], and Choosy [21]. Hadoop [1], [33], [34] divides resources into map/reduce slots and allocates them fairly across pools and jobs. In contrast, YARN [4] partitions resources into containers (i.e., a set of memory and CPU cores) and tries to guarantee fairness among queues. Quincy [24] achieves the fairness of scheduling multiple jobs by formulating it as a min-cost flow problem. Mesos [23] enables multiple diverse computing frameworks such as Hadoop and MPI sharing a single cluster system.

Choosy [21] extends the max-min fairness by considering placement constraints. However, all these schedulers are *memoryless* and thus *not* suitable for resource sharing in the pay-as-you-use computing system. In comparison, Tang et al. [31] proposed a long-term resource fairness (LTRF) policy and developed a prototype LTYARN [31] that implements the LTRF in YARN for the single-resource fairness. They showed that LTYARN is suitable for pay-as-you-go computing system. In contrast, this paper considers the multi-resource fairness allocation with multiple resource types. We have made the following observations and contributions. First, we show that the currently popular multi-resource fair policies such as DRF and AF still have serious fairness problems, due to their *memoryless* feature. Second, we observe the naive extensions of DRF (or AF) with the long-term notion cannot meet all the desired properties. Next, by analyzing two attempts, we propose H-MRF that generalizing DRF and AF with the long-term notion for multi-resource fairness in the pay-as-you-use computing system. Finally, we develop MRYARN, which implements H-MRF in YARN for multi-resource allocation in the pay-as-you-use computing system.

9 Conclusion

Resource sharing is an efficient way to improve the resource utilization of a computing system. Multi-resource fairness is needed and important due to the heterogeneous resource demands for tasks in practice. However, we observe that the popular multi-resource fair policy, DRF, used in existing systems such as YARN, is *not* suitable for contribute-as-you-use computing, due to the untruthfulness problem and resource-as-you-contributed unfairness problem. To address the problems, we have proposed H-MRF and show that it is suitable for contribute-as-you-use computing. Finally, we implement H-MRF in YARN by developing a prototype MRYARN. Our experiments show that 1). all users have sharing benefits under H-MRF, whereas DRF does not; 2). Resource sharing with H-MRF can achieve a higher resource utilization and better performance (about $1.2\times \sim 1.8\times$ performance improvement over static partitioning) than alternative resource sharing policies; 3) users under H-MRF can get the highest monetary cost savings compared to its alternatives (e.g., DRF, LT-DRF, LT-AF).

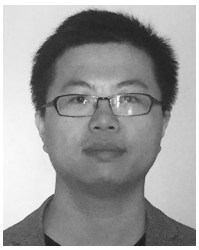
Acknowledgments

This work is sponsored by the National Natural Science Foundation of China (61602336, 61772544, U1731125). Ce Yu's work is supported by the National Natural Science Foundation of China (U1531111, U1731243). Bingsheng and Zhaojie's work is supported by a MoE AcRF Tier 1 grant (T1 251RES1610) and an NUS startup grant in Singapore.

References

[1] Apache hadoop. [Online]. Available: http://hadoop.apache.org, 2017.
[2] Apache hive performance benchmarks. [Online]. Available: https://issues.apache.org/jira/browse/HIVE-396, 2017.
[3] Apache TPC-H benchmark on hive. [Online]. Available: https://issues.apache.org/jira/browse/HIVE-600, 2016.
[4] Apache yarn. [Online]. Available: https://hadoop.apache.org/docs/current2/index.html, 2017.
[5] Blue waters. [Online]. Available: http://www.ncsa.illinois.edu/enabling/bluewaters/, 2017.
[6] Comet. [Online]. Available: http://www.sdsc.edu/services/hpc/hpc_systems.html, 2017.
[7] Facebook workload traces. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository, 2012.
[8] Google cluster data. [Online]. Available: https://code.google.com/p/googleclusterdata/, 2014.
[9] Max-min fairness (Wikipedia). [Online]. Available: http://en.wikipedia.org/wiki/Max-min_fairness, 2017.
[10] Mira. [Online]. Available: http://www.alcf.anl.gov/mira, 2017.
[11] Puma datasets. [Online]. Available: http://web.ics.purdue.edu/~fahmad/datasets.htm, 2014.
[12] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, "Cloud computing pricing models: A survey," *Int. J. Grid Distrib. Comput.*, vol. 6, no. 5, pp. 93–106, 2013.
[13] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "PUMA: Purdue MapReduce benchmarks suite," ECE Technical Reports, 2012. [Online]. Available: https://docs.lib.purdue.edu/ecetr/437
[14] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," in *Proc. ACM Symp. Cloud Comput.*, 2013, pp. 4:1–4:15.
[15] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on HPC systems with node sharing," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2013, pp. 37:1–37:12.
[16] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2004, pp. 10–10.
[17] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 127–144.
[18] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints: On fair sharing of multiple resources," in *Proc. Conf. Innovations Theoretical Comput. Sci.*, 2012, pp. 68–75.
[19] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proc. Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2012, pp. 1–12.
[20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[21] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 365–378.
[22] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2014, pp. 455–466.
[23] B. Hindman, et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM Symp. Operating Syst. Principles*, 2009, pp. 261–276.
[25] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework," *IEEE/ACM Trans. Netw.*, vol. 21, no. 6, pp. 1785–1798, Dec. 2013.
[26] I. Kash, A. D. Procaccia, and N. Shah, "No agent left behind: Dynamic fair division of multiple resources," in *Proc. Int. Conf. Auton. Agents Multiagent Syst.*, 2013, pp. 351–358.
[27] H. Liu and B. He, "Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in IaaS clouds," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2014, pp. 970–981.
[28] D. C. Parkes, A. D. Procaccia, and N. Shah, "Beyond dominant resource fairness: Extensions, limitations, and indivisibilities," *ACM Trans. Econ. Comput.*, vol. 3, no. 1, pp. 3:1–3:22, Mar. 2015.
[29] J. Sahu and K. R. Heavey, "Advanced computational fluid dynamics simulations of proiectiles with flow control," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2004, pp. 27–27.
[30] S. Tang, B. He, S. Zhang, and Z. Niu, "Elastic multi-resource fairness: Balancing fairness and efficiency in coupled CPU-GPU architectures," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2016, pp. 75:1–75:12.

[31] S. Tang, B.-S. Lee, B. He, and H. Liu, "Long-term resource fairness: Towards economic fairness on pay-as-you-use computing systems," in *Proc. Annu. Int. Conf. Supercomput.*, 2014, pp. 251–260.
[32] S. Tang, B.-S. Lee, and B. He, "Fair resource allocation for data-intensive computing in the cloud," *IEEE Trans. Services Comput.*, vol. 11, no. 1, pp. 1–14, 2018.
[33] S. Tang, B.-S. Lee, and B. He, "Dynamic job ordering and slot configurations for MapReduce workloads," *IEEE Trans. Services Comput.*, vol. 9, no. 1, pp. 4–17, Jan. 2016.
[34] S. Tang, B.-S. Lee, and B. He, "DynamicMR: A dynamic slot allocation optimization framework for MapReduce clusters," *IEEE Trans. Cloud Comput.*, vol. 2, no. 3, pp. 333–347, Oct. 2014.
[35] A. Thusoo, et al., "Hive - A petabyte scale data warehouse using hadoop," in *Proc. IEEE Int. Conf. Data Eng.*, Mar. 2010, pp. 996–1005.
[36] W. Wang, B. Li, and B. Liang, "Dominant resource fairness in cloud computing systems with heterogeneous servers," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 583–591.
[37] W. Wang, B. Li, B. Liang, and J. Li, "Multi-resource fair sharing for datacenter jobs with placement constraints," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2016, pp. 86:1–86:12.
[38] P. Webster, "NASA center for climate simulation: Data-centric climate computing, in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2011, pp. 1–6.
[39] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[41] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 29–42.
[42] S. M. Zahedi and B. C. Lee, "REF: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 145–160.



Shanjiang Tang received the BS and MS degrees from Tianjin University, China, in January 2008 and July 2011, respectively, and the PhD degree from the School of Computer Science and 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.

Zhaojie Niu received the bachelor's and master's degrees from the Huazhong University of Science and Technology (HUST), China, in January 2009 and July 2012 respectively, and the PhD degree from Nanyang Technological University (NTU), Singapore, in September 2017. He is a research associate with the National University of Singapore (NUS). He is interested in resource management, job scheduling, and data processing in large-scale clusters.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University, in 2003 and the PhD degree in computer science from the Hong Kong University of Science and Technology, in 2008. He is an associate professor in the School of Computing, National University of Singapore. His research interests include high performance computing, distributed and parallel systems, and database systems.

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 in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research interests include computer networks protocols, distributed computing, network management, and Grid/Cloud computing.

Ce Yu received the BS and MS degrees from Tianjin University, in 1998 and 2005, respectively, and the PhD degree in computer science from Tianjin University (TJU), in 2009. He is currently an associate professor and director of High Performance Computing Lab (HPCL) of Computer Science & Technology in Tianjin University. His main research interests include high performance computing, big data, astro-informatics, and cloud computing.