9 Resource Management in Big Data Processing Systems

Shanjiang Tang, Bingsheng He, Haikun Liu and Bu-Sung Lee

Contents

Abstract

Resource management is a fundamental design issue for big data processing systems in the cloud. Different resource allocation policies can have significantly different impacts on performance and fairness. In this chapter, we first make an overview of existing big data processing and resource management systems. Next we have a study on the economic fairness for large-scale resource management on the cloud according to some desirable properties including sharing incentive, truthfulness, resource-as-you-pay fairness, and pareto efficiency. Both single-resource and multi-resource management are studied for cloud computing. We show that the proposed resource allocation policies can meet all desired properties and achieve good performance results. Lastly, some open questions are also proposed and discussed.

Key Words: Resource Management, Cloud Computing, Single-resource, Multi-resource, Hadoop, Spark, Pay-as-you-go, Resource Sharing, Fairness, Performance, Big Data.

9.1. Introduction

In many application domains such as social networks and Bio-informatics, the data is being gathered at unprecedented scale. Efficient processing for "big data" analysis poses new challenges for almost all aspects of state-of-the-art data processing and management systems. For example, there are a few challenges as follows: (i) the data can be arbitrarily complex structures (e.g., graph data) and cannot be efficiently stored in relational database; (ii) the data access of large-scale data processing are frequent and complex, resulting in inefficient disk I/O accesses or network communications; and (iii) last but not least, to tackle a variety of unpredictable failure problems in the distributed environment, data processing system must have a fault tolerance mechanism to recovery the task computation automatically.

Cloud computing has emerged as an appealing paradigm for big data processing over the Internet due to its cost effectiveness and powerful computational capacity. Current Infrastructure-as-a-Service(IaaS) clouds allow tenants to acquire and release resource in the form of virtual machines (VMs) on a pay-as-you-go basis. Most IaaS cloud providers such as Amazon EC2 offer a number of VM types (such as *small*, *medium*, *large* and *extra large*) with fixed amount of CPU, main memory and disk. Tenants can only purchase fixed-size VMs and increase/decrease the number of VMs when the resource demands change. This is known as *T-shirt and scale-out model* [32]. However, the T-shirt model leads to inefficient allocation of cloud resource, which translates to higher capital expense and operating cost for cloud providers, and increase of monetary cost for tenants. First, the granularity of resource acquisition/release is coarse in the sense that the fix-sized VMs are not tailored for cloud applications with dynamic demands delicately. As a result, tenants need to over-provision resource (costly), or risk performance penalty and Service Level Agreement (SLA) violation. Second, elastic resource scaling in clouds [13], also known as scale-out model, is also costly due to the latencies involved in VM instantiating [55] and software runtime overhead [56]. These costs are ultimately borne by tenants in terms of monetary cost or performance penalty.

Resource sharing is a classic and effective approach to resource efficiency. As more and more big data applications with diversifying and heterogeneous resource requirements tend to deploy in the cloud [10][34], there are vast opportunities for resource sharing [30][32]. Recent work has shown that fine-grained and dynamic resource allocation techniques (e.g., resource multiplexing or overcommitting [14][19][32][39][67]) can significantly improve the resource utilization compared to T-shirt model [32]. As adding/removing resource is directly performed on the existing VMs, fine-grained resource allocation is also known as scale-up model and the cost tends to much smaller compared to the scale-out model. Unfortunately, current IaaS clouds do not offer resource sharing among VMs, even if those VMs belong to the same tenant. Resource sharing models are needed for better cost efficiency of tenants and higher resource utilization of cloud providers.

Researchers have been actively proposing many innovative solutions to address the new challenges of large scale data processing and resource management. In particular, a notable number of large-scale data processing and resource management systems have recently been proposed. The aims of this chapter are (i) to introduce canonical examples of large data processing, (ii) to make an overview of existing data processing and resource management systems/platforms, and more importantly, (iii) to make a study on the economic fairness for large-scale resource management on the cloud, which bridges large-scale resource management and cloud-based platforms. In particular, we present some desirable properties including sharing incentive, truthfulness, resource-as-you-pay fairness and pareto efficiency to guide the design of fair policy for the cloud environment.

The chapter is organized as follows. We first present several types of resource management for big data processing in Section 1. In Section 9.3, we list some representative big data processing platforms. Section 9.4 presents the single resource management in the cloud, following by Section 9.5 that introduces multi-resource management in the cloud. For the completeness of discussions, Section 9.6 gives an introduction to existing work on resource management. A discussion of open problems is provided in Section 9.7. We conclude the chapter in Section 9.8.

9.2. Types of Resource Management

Resource management is a general and fundamental issue in computing systems. In this section, we present the resource management for typical resources including CPU, memory, storage and network.

9.2.1. CPU and Memory Resource Management

Current supercomputers and data centers (e.g., Amazon EC2) generally consist of thousands of computing machines. At any time, there are tens of thousands of users running their high-performance computing applications (e.g., MapReduce [23], MPI, Spark [76]) on it. The efficient resource management of the computing resources such as cpu, memory is non-trivial for high performance and fairness. Typically, the resource management includes resource discovery, resource scheduling, resource allocation and resource monitoring. Resource discovery identifies the suitable computing resources in which machines that match the user's request. Resource scheduling selects the best resource from the matched computing resources. It actually identifies the resource where the machines are to be created to provision the resources. Resource allocation allocates the selected resource to the job or task of user's request. Actually, it means the job submission to the selected cloud resource. After the submission of the job, the resource is monitored.

There are a number of resource management tools available for supercomputing. For example, SLURM is a highly scalable resource manager widely used in supercomputers [74]. It allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work. For data-intensive computing in data center, YARN [65] and Mesos [40] are two popular resource management systems.

9.2.2. Storage Resource Management

Storage resource management (SRM) is a proactive approach to optimizing the efficiency and speed with which available drive space is utilized in a storage area network, which is a dedicated high-speed network (or subnetwork) that interconnects and presents shared pools of storage devices to multiple servers [6]. The SRM software can help a storage administrator automate data backup, data recovery and SAN performance analysis. It can also help the administrator with configuration management and performance monitoring, forecast future storage needs more accurately and understand where and how to use tiered storage, storage pools and thin provisioning.

9.2.3. Network Resource Management

Managing and allocating the network flows to different applications/users is a non-trivial work. Particularly, Software-defined networking (SDN) [5] is nowadays a popular approach that allows network administrators to manage network services through abstraction of lower-level functionality. This is done by decoupling the system that makes decisions about where traffic is sent (the control plane) from the underlying systems that forward traffic to the selected destination (the data plane).

9.3. Big Data Processing Systems and Platforms

In the era of Big Data, characterized by the unprecedented volume of data, the velocity of data generation, and the variety of the structure of data, support for large-scale data analytics constitutes a particularly challenging task. To address the scalability requirements of todays data analytics, parallel shared-nothing architectures of commodity machines (often consisting of thousands of nodes) have been lately established as the de-facto solution. Various systems have been developed mainly by the industry to support Big Data analysis, including MapReduce [23], Pregel [54], Spark [76], etc. In this section, we give a brief survey of some representative solutions.

9.3.1. Hadoop

Hadoop [2] is an open-source java implementation of MapReduce [23], which is a popular programming model for large scale data processing proposed by Google. It runs on top of a distributed file system called HDFS, which splits an input data into multiple blocks of fixed size (typically 64MB) and replicates each data block several times across computing nodes. Users can submit MapReduce jobs to the Hadoop cluster. The Hadoop system breaks each job into multiple map tasks and reduce tasks, with its map tasks computed before its reduce tasks. Each map task processes (i.e. scans and records) a data block and produces intermediate results in the form of key-value pairs.

Moreover, Hadoop has evolved to the next generation of Hadoop (i.e., Hadoop MRv2) called YARN [65], as a large-scale data operating platform and cluster resource management system. There is a new architecture for YARN, which separates the resource management from the computation model. Such a separation enables YARN to support a number of diverse data-intensive computing frameworks including Dryad [42] , Giraph [1] , Spark [76]
, Storm [7] and Tez [8]. In YARNs architecture, there is a global master named ResourceManager (RM) and a set of per-node slaves called NodeManagers (NM), which forms a generic system for managing applications in a distributed manner. The RM is responsible for tracking and arbitrating resources among applications. In contrast, the NM has responsibility for launching tasks and monitoring the resource usage per slave node. Moreover, there is another component called ApplicationMaster (AM), which is a framework-specific entity. It is responsible for negotiating resources from the RM and working with the NM to execute and monitor the progress of tasks. Particularly, all resources of YARN are requested in the form of container, which is a logical bundle of resources (e.g., ă1 CPUs, 2G memoryą). As a multi-tenant platform, YARN organizes users submitted applications into queues and share resources between these queues. Users can set their own queues in a configuration file provided by YARN.

9.3.2. Dryad

Dryad [42] is a distributed execution engine that simplifies the process of implementing data-parallel applications to run on a cluster. The original motivation for Dryad was to execute data mining operations efficiently, which has also lead to technologies such as MapReduce or Hadoop. However, Dryad is a general-purpose execution engine and can also be used to implement a wide range of other application types, including time series analysis, image processing, and a variety of scientific computations. The computation is structured as a directed graph: programs are graph vertices, while the channels are graph edges. A Dryad job is a graph generator which can synthesize any directed acyclic graph. These graphs can even change during execution, in response to important events in the computation. Dryad handles job creation and management, resource management, job monitoring and visualization, fault tolerance, re-execution, scheduling, and accounting.

9.3.3. Pregel

Pregel [54] is a specialized model for iterative graph applications. In Pregel, a program runs as a series of coordinated supersteps. On each superstep, each vertex in the graph runs a user function that can update state associated with the vertex, change the graph topology, and send messages to other vertices for use in the next superstep. This model can express many graph algorithms, including shortest paths, bipartite matching, and PageRank.

9.3.4. Storm

Storm [7] is a distributed real-time computation system. Similar to how Hadoop provides a set of general primitives for doing batch processing, Storm provides a set of general primitives for doing real-time computation, which greatly ease the writing of parallel real-time computation. It can be used for processing messages and updating databases (stream processing), doing a continuous query on data streams and streaming the results into clients (continuous computation), parallelizing an intense query like a search query on the fly (distributed RPC), and more. Storm's small set of primitives satisfy a stunning number of use cases. Storm scales to massive numbers of messages per second. To scale a topology, all you have to do is add machines and increase the parallelism settings of the topology. As an example of Storm's scale, one of Storm's initial applications processed 1,000,000 messages per second on a 10 node cluster, including hundreds of database calls per second as part of the topology.

9.3.5. Spark

Spark [76] is a fast in-memory data processing system that achieves high performance for applications through caching data in memory (or disk) for data sharing across computation stages. It is achieved by proposing *Resilient Distributed Dataset (RDD)* in-memory storage abstraction for computing data, which is a read-only, partitioned collection of records [75]. Each RDD is generated from data in stable storage or other RDDs through *transformation* operations such as *map*, *filter* and *reduceByKey*. Notably, the *transformation* is a *lazy* operation that only defines a new RDD without immediately computing it. To launch RDD computation, Spark provides another set of *action* operations such as *count*, *collect* and *save*, which return a value to the application or export data to a storage system. For RDD caching, Spark offers five storage levels, i.e., *MEMORY_ONLY*, *MEMORY_AND_DISK*, *MEMORY_ONLY_SER*, *MEMORY_AND_DISK_SER* and *DISK_ONLY*. For a Spark application in execution, the Spark system will spawn a master called *driver* responsible for defining and managing RDDs and a set of slavers called *executors* to do the computation dynamically. The Spark applications can run on either YARN, Mesos, local, or standalone cluster modes. In this paper, we focus on the standalone cluster mode.

9.3.6. Summary

In this section, we summarize some representative big data processing systems elaborated in the previous section in terms of computation model, in-memory computation, resource management type (i.e., cpu, memory, storage and network) and fairness in Table 9.1.

There are some other high-level as well as application-specific systems that are built on top of previous data computing systems to form an ecosystem for a variety of applications. For example, for Hadoop, Apache Pig [57] and Hive [64] are both SQL-like systems that are running on it to support analytical data querying processing. HBase [29] is a NoSql database system built on top of Hadoop system. Apache Giraph [1] is an iterative graph pro- cessing system running on Hadoop. Similarly, for Spark, Shark [73] and Spark SQL [12] are two analytical data query system built on Spark, and Graphx [33] is a graph processing system for graph applications. We have also witnessed some other data processing sys- tems/platforms that are running on currently emerging computing devices such as GPUs. For example, as an extension of Pregel for GPU platform, a general-purpose programming framework called Medusa [77] has been developed for graph applications.

Systems	Computation Model	In-memory computation	Resource Management Type				Resource Fairness	
			CPU	Memory	Storage	Network	Single	Multiple
Hadoop	MapReduce	no	yes	yes	no	no	yes	yes
Dryad	Dryad	no	yes	no	no	no	yes	no
Pregel	Pregel	no	yes	no	no	no	yes	no
Storm	Storm	no	yes	no	no	no	yes	no
Spark	RDD	yes	yes	yes	no	no	yes	yes

Table 9.1: Comparison of representative big data processing systems.

9.4. Single Resource Management in the Cloud

Single resource management refers to the management of single resource type. This is the basic form of resource management. For example, Hadoop manages CPU resources of a computing cluster in the form of slots.

Cloud computing has emerged as a popular platform for users to compute their large- scale data applications, attracting from its merits such as flexibility, elasticity, and cost efficiency. At any time, there can be ten to thousands of users concurrently running their large-scale data-intensive applications on the cloud. Users pay the money on the basis of their resource usage. To meet different users needs, cloud providers generally offer several options of price plans (e.g., on-demand price, reserved price). When a user has a short- term computation requirement (e.g., one week), 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 on the hourly charge for the resources in comparison to the on-demand one and thereby save the money cost.

To improve the resource utilization and in turn the cost saving, resource sharing is an effective approach [30]. Consider the reserved resources for example. With reservation plan, users need to pay a one-time fee for a long time (e.g., 1 or 3 years), and in turn get a significant discount on the hourly usage charge. However, to achieve the full cost savings, users must commit to have a high utilization. In practice, the resource demand of a user can fluctuate over time, and the resources cannot be fully utilized all the time from the perspective of an individual user. With resource sharing, users can complement from each other in the resource usage in a shared cluster and the resource utilization problem can be thereby solved. To make resource sharing possible among users, the resource allocation fairness is a key issue.

As shown in Table 9.1, there are single and multiple resource fairness, both of which are supported by some of computing systems. In this section, we mainly focus on the fairness of single resource management, and defer the fairness for multiple resource management to Section 9.5. Notably, 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 Re- source Fairness (MLRF)*. In the following subsections, we first present several desirable resource allocation properties for cloud computing. Next we show the problems for MLRF. Finally, we explore a new policy to address it.

9.4.1. Desired Resource Allocation Properties

This section presents a set of desirable properties that we believe any cloud-oriented re-source allocation policy in a shared pay-as-you-use system should meet.

- *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/n resources in a system partition consisting of $1/n$ 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 possesses 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.

9.4.2. Problems for Existing Fairness Policies

One of the most popular fair allocation policy is *(weighted) max-min fairness [30]*, which maximizes the minimum resource allocation obtained by a user in a shared computing sys- tem. It has been widely used in many popular large-scale data processing frameworks such as Hadoop [2], YARN [65], Mesos [40], and Dryad [42]. 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 cloud 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 9.4.1). 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 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 9.4.1).

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 Sec- tion 1.4.1). 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 hap- pen 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 Unfairness 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 9.4.1). 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.

9.4.3. Long-Term Resource Allocation Policy

[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.

In this section, we first give a motivation example to show that MemoryLess Resource Fairness (MLRF) is *not* suitable cloud computing system. Then we propose Long-Term Resource Fairness (LTRF), a cloud-oriented allocation policy to address the limitations of MLRF and meet the desired properties described in Section 9.4.1.

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 9.2, 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 9.2(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 9.2(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*' [3] 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 9.2(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 9.2(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*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 * 50%). Still, both of them are *fair* at time t_4.

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.*/
10:        Update the total resource usage $u_i$ for client $i$.
11:        Allocate resource to client $i$.
12:    **else**
13:        /*The system is fully utilized.*/
14:        **Wait** until there is a released resource $r_i$ from client $i$.
15:        $\ddot{R}_i \leftarrow \ddot{R}_i - r_i$. /*Update current allocated resources */
```

9.4.4. Experimental 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

	Client A					Client B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	ˊ30	100	100	80	80	ˋ30
t_2	40	40	40	60	ˊ10	60	80	60	140	ˋ10
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.

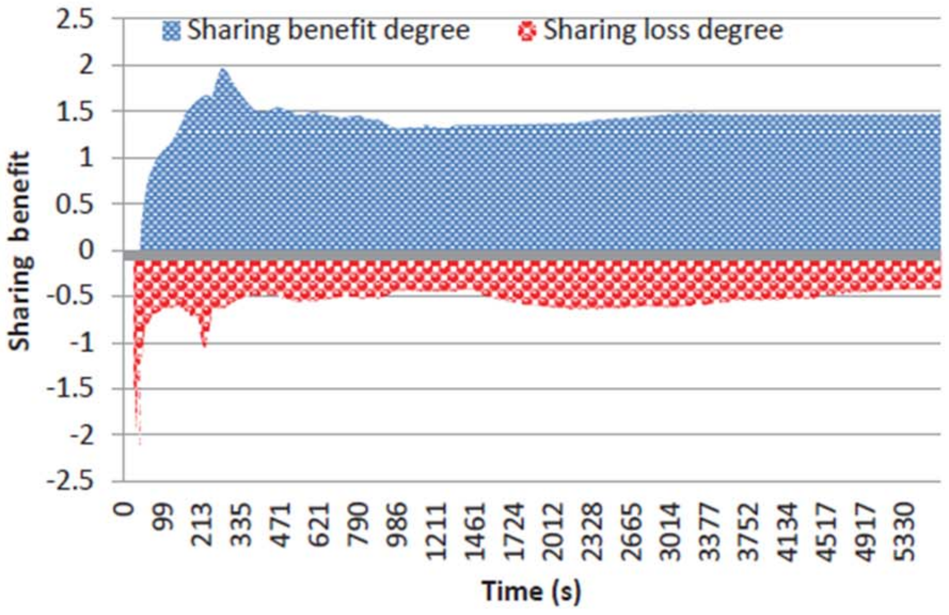
	Client A					Client B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	ˊ30	100	100	80	80	ˋ30
t_2	40	40	40	60	ˊ10	60	80	60	140	ˋ10
t_3	80	80	80	140	ˋ30	50	70	20	160	ˊ30
t_4	60	60	60	**200**	ˋ10	50	100	40	**200**	ˊ10

(b) Allocation results based on *LTRF*.

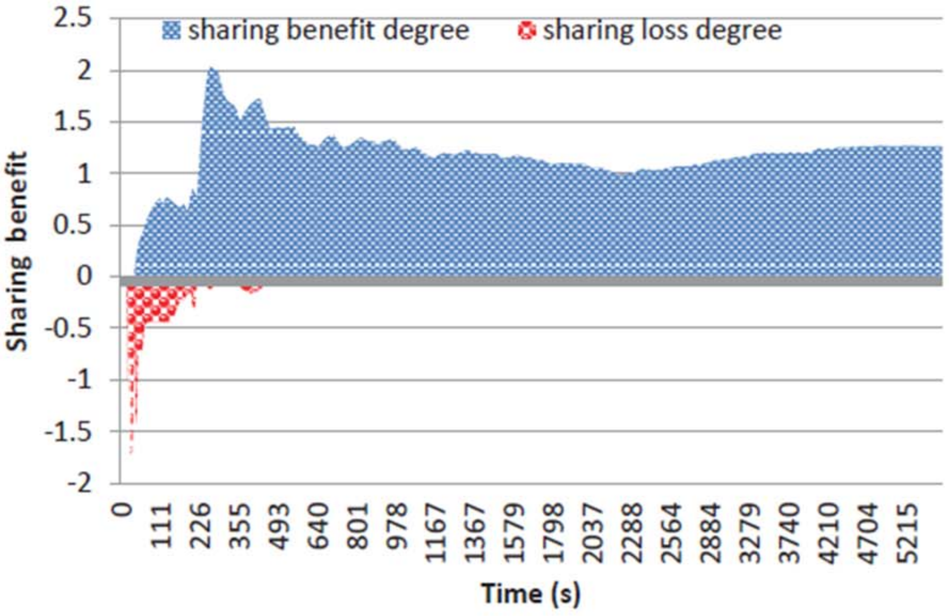
	Client A					Client B				
	Demand		Counted Allocation			Demand		Counted Allocation		
	New	Total	Current	Total	Preempt	New	Total	Current	Total	Preempt
t_1	20	20	20	20	ˊ30	100	100	65	65	ˋ30
t_2	40	40	40	60	ˊ10	60	80	55	120	ˋ10
t_3	80	80	65	125	ˋ30	50	70	20	140	ˊ30
t_4	60	60	55	**180**	ˋ10	50	100	40	**180**	ˊ10

(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 9.2(b).

Table 9.2: 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.



(a) Sharing benefit/loss degree with MLRF.



(b) Sharing benefit/loss degree with LTRF.

Figure 9.1: Comparison of fairness results over time for workloads under MLRF and LTRF in YARN. All results are relative to the static partition scenario (i.e., non-shared case) whose sharing benefit/loss is zero. (a) and (b) show the overall benefit/loss relative to the non-sharing scenario.

nodes). For each slave node, we configure its total memory resources with 24GB. The second level denotes the applications (i.e., workloads). We ran a macro-benchmark consisting of four different workloads. Thus, four different queues are configured in YARN/LTYARN, 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 [27]. 3). Hive running a series of TPC-H queries. 4). Spark running a series of machine learning applications.

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 9.1. All results are relative to the static partition case (without sharing) with sharing benefit/loss degrees of zero. Figures 9.1(a) and 9.1(b) present the sharing benefit/loss degrees, respectively, for MLRF and LTRF. The following observations can be obtained: First, the sharing policies of both MLRF and LTRF can bring sharing benefits for queues (workloads). 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 9.1(a) indicates that the sharing loss problem for MLRF is constantly available until all the workloads complete (e.g., about -0.5 on average), 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. 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 ~ 650 seconds) of LTRF in Figure 9.1(b), 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.

9.5. Multi-Resource Management in the Cloud

Despite the resource sharing opportunities in the cloud, resource sharing, especially for *multiple* resource types (i.e., multi-resource), poses several important and challenging problems in pay-as-you-use commercial clouds. (1) *Sharing incentive*. In a shared cloud, a tenant may have concerns about the gain/loss of her asset in terms of resource. (2) *Free riding*. A tenant may deliberately buy less resource than her demand and always expect to benefit from others' contribution (i.e., unused resource). Free Riders would seriously hurt other tenants' sharing incentive. (3) *Lying*. When there exists resource contention, a tenant may lie about her resource demand for more benefit. Lying also hurts tenants' sharing incentive. (4) *Gain-as-you-contribute Fairness*. It is important to guarantee that the allocations obey a rule "more contribution, more gain". In summary, those problems are eventually attributed to economic fairness of resource sharing in IaaS clouds. Unfortunately, the popular allocation policies such as (Weighted) Max-Min Fairness (WMMF) [46] and Dominant Resource Fairness (DRF) [30] cannot address all the problems of resource sharing in IaaS clouds (see Section **Error! Reference source not found.**).

This Section introduce F2C, a cooperative resource management system for IaaS clouds. F2C exploits statistical resource complementarity to group tenants together to realize the resource sharing opportunities, and adopts a novel resource allocation policy to guarantee fairness of resource sharing. Particularly, we describe *Reciprocal Resource Fairness (RRF)*, a generalization of max-min fairness to multiple resource types [50]. The intuition behind RRF is that each tenant should preserve her asset while maximizing the resource usage in a cooperative environment. Different types of resource is advocated to trade among different tenants and to share among different VMs belonging to the same tenant. For example, a tenant can trade her unused CPU share for other tenant's over-provisioned memory share. In this Section, we consider two major kinds of resources, including CPU and main memory. Resource trading can maximize tenants' benefit from resource sharing. RRF decomposes the multi-resource fair sharing problem into a combination of two complementary mechanisms: Inter-tenant Resource Trading (IRT) and Intra-tenant Weight Adjustment(IWA). These mechanisms guarantee that tenants only allocate minimum shares to their non-dominant demands and maximize the share allocations on the contended resource. Moreover, RRF is able to achieve some desirable properties of resource sharing, including sharing incentive, gain-as-you-contribute fairness and strategy-proofness(guarding against free-riding and lying).

In the following, we first introduce the resource allocation model in F2C. Second, we analysis the problems of two popular resource allocation policies including Weighted Max-Min Fairness (WMMF) [46] and Dominant Resource Fairness (DRF) [30]. Third, we present RRF, the resource allocation model in F2C. Finally, we evaluate F2C and show how RRF can address resource fair allocation problems in IaaS clouds.

9.5.1. Resource Allocation Model

We consider the resource sharing model in multi-tenant cloud environments, where each tenant may rent several VMs to host her applications, and VMs have multi-resource demands. By *multi-resource*, we mean resource of *different resource types*, instead of multiple units of the same resource type. In this paper, we mainly consider two resource types: CPU and main memory. Tenants can have different *weights* (or *shares*) to the resource. The *share* of a tenant reflects the tenant's priority relative to other tenants. A number of tenants can form a resource pool based on the opportunities of resource sharing. The VMs of these tenants then share the same resource pool with negotiated resource *shares*, which are determined by tenants' payment.

In our resource allocation model, each unit of resource is represented by a number of shares. To simplify multi-resource allocation, we assume each unit of resource (such as 1 Compute Unit [1] or 1 GB RAM) has its fixed share according to its market price. A study on Amazon EC2 pricing data [72] had indicated that the hourly unit cost for 1 GB memory is twice as expensive as one EC2 Compute Unit. A tenant's *asset* is then defined as the aggregate shares that she pays for.

[1]One EC2 Compute Unit provides equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, according to Amazon EC2.

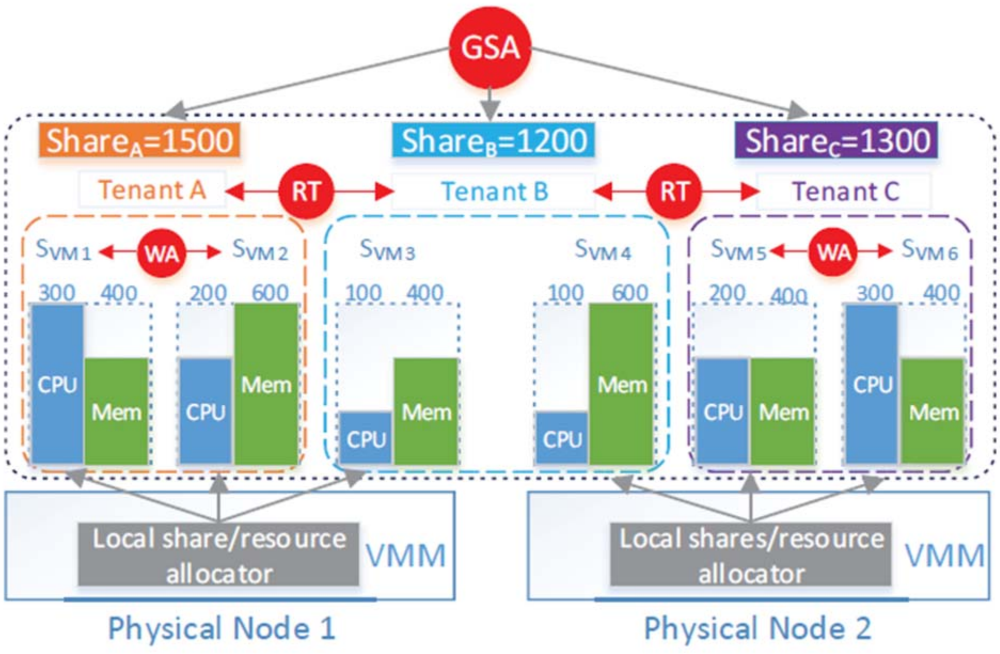


Figure 9.2: Hierarchical resource allocation based on resource trading and weight adjustment.

In the following, we use an example to demonstrate our resource allocation model. Figure 9.2 shows three tenants co-located on two physical hosts. Each tenant has two VMs. The shares of different resources (e.g., CPU, Memory) are uniformly normalized based on their market prices [72]. To some extent, a tenant actually purchases resource shares instead of fix-sized resource capacity. As share of a VM reflects the VMs priority relative to other VMs, cloud providers can directly use shares as billing and resource allocation policies. The concrete design of those policies is beyond the scope of this paper. Thus, we simply define a function f_1 to translate tenants' payment into shares $payment \xrightarrow{f_1} share$, and another function f_2 to translate shares into resource capacity $share \xrightarrow{f_2} resource$. For example, in Figure 9.2, one compute unit and one GB memory are priced at 100 and 200 shares, respectively. If VM1 is initialized with 3 compute units and 2 GB memory, VM1 is allocated with total $100 \times 3 + 200 \times 2 = 700$ shares.

This model enables fine-grained resource allocation based on shares, and thus provides the opportunities for dynamic resource sharing. Now, the fairness has become a major concern in such a shared system. Informally, we define a kind of *economic fairness*: each tenant should try to maximize her aggregate multi-resource shares if she has unsatisfied resource demand

We find that resource trading between different tenants is able to reinforce the economic fairness of resource sharing. Normalizing multiple resources with uniform share provides advantages to facilitate resource trading. For example, one tenant can trade her over-provisioned CPU shares for other tenants' underutilized memory shares. In Figure 9.2, VM1 may trade its 200 CPU shares for VM3's 100 memory shares. Resource trading can prevent tenants from losing underutilized resource. Moreover, a tenant can dynamically adjust share allocation of her VMs based on the actual demands. For example, in Figure 9.2, tenant A may deprive 200 memory shares from VM2 and re-allocate them to VM1. In this paper, we propose resource trading between different tenants (Section 9.5.3.1), and dynamic weight adjustment among multiple VMs belonging to the same tenant (Section 9.5.3.2). Figure 9.2 shows the hierarchical resource allocation based on these two mechanisms. The global share allocator (GSA) first reserves capacity in bulk based on tenants' aggregate resource demands, and then allocates shares to tenants according to their payment. The local share/resource allocator in each node is responsible for Resource Trading (RT) between tenants, and Weight Adjustment (WA) among multiple VMs belonging to the same tenant.

9.5.2. Multi-resource Fair Sharing Issues

In the following, we demonstrate the deficiency of WMMF and DRF for multi-resource sharing in clouds.

Example 1: Assume there are three tenants, each of which has one VM. All VMs share a resource pool consisting of total 20 GHz CPU and 10 GB RAM. Each VM has initial shares for different types of resource when it is created. For example, VM1 initially has CPU and RAM shares of 500 each, simply denoted by a vector x500*,* 500y. The VMs may have dynamic resource demands. At a time, VM1 runs jobs with demands of 6 GHz CPU and 3 GB RAM, simply denoted by a vector x6$GHz,$ 3$GB$y. The VMs' initial shares and demand vectors are illustrated in Table 9.3. We examine whether T-shirt model, WMMF and DRF can achieve resource efficiency and economic fairness.

With T-shirt Model, we allocate the total resources to tenants in proportion to their share values of CPU and memory separately. The T-shirt model guarantees that each tenant precisely receives the resource shares that the tenant pays for. However, it wastes scare resource because it may over-allocate resource to VMs that has high shares but low demand, even other VMs have unsatisfied demand. As shown in Table 9.3, VM2 wastes 1.5 GB RAM and VM3 wastes 2GHz CPU.

We now apply the WMMF algorithm on each resource type. As shown in Table 9.3, VM1, VM2 and VM3 initially owns 25%, 25%, 50% of total resource shares, respectively. However, VM1 is allocated with 30% of total resources, with 5% "stolen" from other VMs. Ironically, VM2 contributes 1.5 GB RAM and VM3 contributes 2 GHz CPU to other ten- ants. However, they do not benefit more than VM1 from resource sharing because CPU and memory resource are allocated separately. In this case, if VM1 deliberately provisions less resource than its demand and always reckons on others' contribution, then VM1 becomes a free rider. Although WMMF can guarantee resource efficiency, it cannot fully preserve tenant's resource shares, and eventually results in economic unfairness.

Table 9.3: Comparison of resource allocation polices between T-shirt, WMMF and DRF.

VMs	VM1	VM2	VM3	Total
Initial Shares	<500, 500>	<500, 500>	<1000, 1000>	<2000, 2000>
Demands	<GHz, 3 GB>	<8 GHz, 1 GB>	<8 GHz, 8 GB>	<22 GHz, 12 GB>
T-shirt Allocation	<5 GHz, 2.5 GB>	<5 GHz, 2.5 GB>	<10 GHz, 5 GB>	<18 GHz, 8.5 GB>
WMMF Allocation	<6 GHz, 3 GB>	<6 GHz, 1 GB>	<8 GHz, 6 GB>	<20 GHz, 10 GB>
WDRF dominant share	6/20 = 3/10	8/20 CPU	8/(10*2) RAM	100%
WDRF Allocation	<6 GHz, 3 GB>	<7 GHz, 1 GB>	<7 GHz, 6 GB>	<20 GHz, 10 GB>

We also apply weighted DRF (WDRF) [30] to this example. Both CPU and RAM shares of VM1, VM2 and VM3 correspond to a ratio of 1 : 1 : 2. VM1's dominant share can be CPU or memory, both equal to 6/20. VM2's dominant share is CPU share as *max*(8/20, 1/10)=8/20. For VM3, its un-weighted dominant share is memory share 8/10. Its weight is twice of that of VM1 and VM2, so its weighted dominant share is 8/(10*2)=8/20. Thus, the ascending order of three VM's dominant shares is VM1 < VM2 = VM3. According to WDRF, VM1's demand is first satisfied, and then the remanding resources are allocated to VM2 and VM3 based on max-min fairness. We find that VM1 is again a free rider.

In summary, the T-shirt model is not resource-efficient, and WMMF and DRF are not economically fair for multi-resource allocation. Intuitively, in a cooperative environment, the more one contributes, the more she should gain. Otherwise, the tenants would lose their sharing incentives. This is especially important for resource sharing among multiple tenants in pay-as-you-use clouds. Thereby, a new mechanism is needed to reinforce the fairness of multi-resource sharing in IaaS clouds.

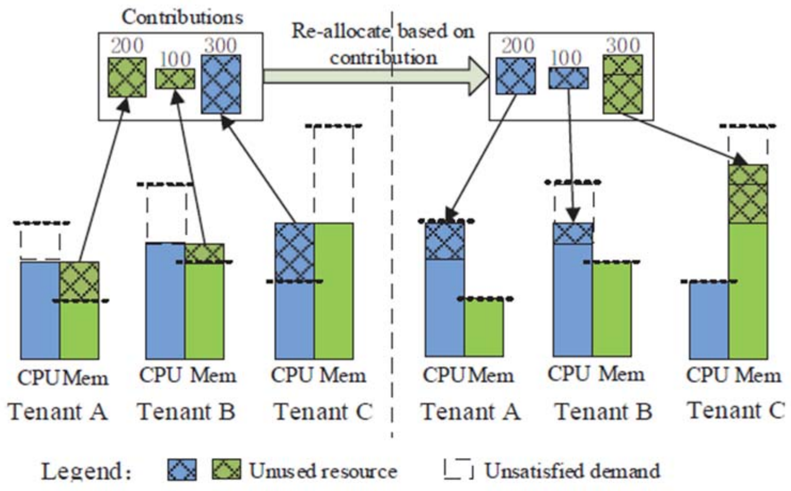


Figure 9.3: Sketch of inter-tenant resource trading.

9.5.3. Reciprocal Resource Fairness

In the following, we consider the fair sharing model in a shared system with p types of resource and m tenants. The total system capacity bought by the m tenants is denoted by a vector Ω, i.e., $<\Omega_{CPU}, \Omega_{RAM}>$, denoting the amount of CPU and memory resource, respectively. Each tenant i may have n VMs. Each VM j is initially allocated with a share vector $s(j)$ that reflects its priority relative to other VMs. The amount of resource share required by VM j is characterized by a demand vector $d(j)$. Correspondingly, the resource share lent to other tenants becomes $s(j) - d(j)$, and we call it a contribution vector $c(j)$. At last, let $s^1(j)$ denote the current share vector when resources are re-allocated. For simplicity, we assume that resource allocation is oblivious, meaning that the current allocation is not affected by previous allocations. Thus, a VM's priority is always determined by its initial share vector $s(j)$ in each time of resource allocation.

9.5.3.1. Inter-tenant Resource Trading (IRT)

For multi-resource allocation, it is hard to guarantee that the demands of all resource types are nicely satisfied without waste. For example, a tenant's aggregate CPU demand may be less than her initial CPU share, but memory demand exceeds her current allocation. In this case, she may expect to trade her CPU resource with other tenants' memory resource. Thus, the question is how to trade resources of different types among tenants while guaranteeing economic fairness. RRF embraces an IRT mechanism with the core idea that a tenant's gain from other tenants should be proportional to her contribution. The only basis for underutilized resource allocation is the tenant's contribution, rather than her initial resource share or unsatisfied demand. As shown in Figure 9.3, the memory resource contributed by Tenant A is twice more than that of Tenant B, and Tenant A should receive twice more unused CPU resource (contributed by Tenant C) than Tenant B at first. Then, we need to check whether the CPU resource of Tenant A is over-provisioned. If so, the unused portion should be re-distributed to other tenants. This process should be *iteratively* performed by all tenants because each time of resource distribution may affect other tenants' allocations. While this naive approach works, it can cause unacceptable computation overhead. We further propose a work backward strategy to speed up the unused resource distribution.

For each type of resource, we divide the tenants into three categories: contributors, beneficiaries whose demands are satisfied, and beneficiaries whose demands are unsatisfied, as shown in Figure 9.4. Tenants in the first two categories are *directly* allocated with their demands exactly, and tenants in the third category are allocated with their initial share plus a portion of contributions from the first category. However, A challenging problem is how to divide the tenants into three categories efficiently. Algorithm 2 describes the sorting process by using some heuristics.

Let vectors $D(i)$, $S(i)$, $C(i)$ and $S^1(i)$ denote the total demand, initial share, contribution and current share of the tenant i, respectively. Correspondingly, let $D_k(i)$, $S_k(i)$, $C_k(i)$ and $S_k^1(i)$ denote her total demand, initial share, contribution and current share of resource type k, respectively. We consider a scenario where m tenants share a resource pool with capacity Ω, with resource contentions (i.e., $\sum_{i=1}^{m} D_i(i) \geq \Omega$). Our algorithm first divide the total capacity on the basis of each tenant's initial share, and then caps each tenant's allocation at her total demand. Actually, each tenant will receive her initial total share Spq, and then her total contribution becomes $C(i) = S(i) - D(i)$ (if $S(i) > D(i)$). For resource type k ($1 \leq k \leq p$), the unused resource $C_k(i)$ is re-distributed to other unsatisfied tenants in the ratio of their total contributions.

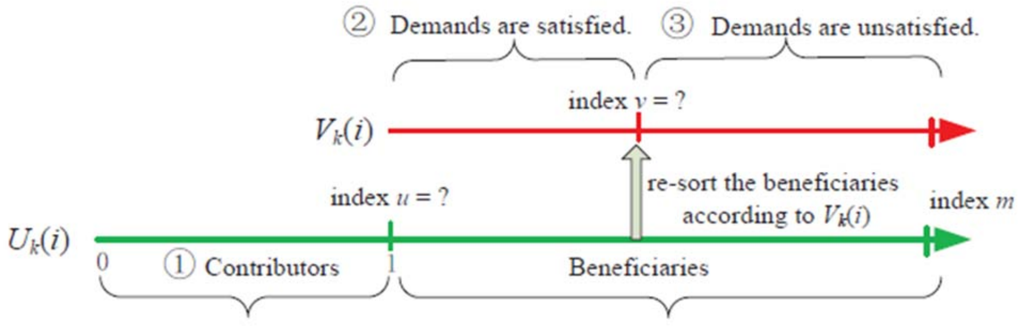


Figure 9.4: Sketch of IRT algorithm.

Algorithm 2 shows the pseudo-code for IRT. We first calculate each tenant's total contribution $\Lambda(i)$ (Lines 6-8). To reduce the complexity of resource allocation, for each resource type k, we define the normalized demand of tenant i as $U_k(i) = D_k(i) / S_k(i)$, and re-index the tenants so that the U_k(i) are in the ascending order, as shown in Figure 9.4. Then, we can easily find the index u so that $U_k(u) < 1$ and $U_k(u+1) \geq 1$. The tenants with index $[1, \ldots, u]$ are contributors and the remaining are beneficiaries. For tenants with index $[u+1, \ldots, m]$ (U_k(i) ≥ 1), we define the ratio of unsatisfied demand of resource type k to her total contribution as $V_k(\mathrm{i}) = (D_k(\mathrm{i}) - S_k(\mathrm{i})) / \sum_{k=1}^{p} C_k(i)$ (Lines 12-13), and re-index these tenants according to the ascending order of V_k(i), as shown in Figure 9.4. Thus, tenants with index $[1, \ldots, u]$ are ordered by U_k(i) while tenants with index $[u+1, \ldots, m]$ are ordered by V_k(i). The demand of tenants with the largest index will be satisfied at last. We need to find a pair of successive indexes v, $v+1$, ($v \geq u$), so that the share allocations of tenants with index [1,… , v] are capped at their demands, and the remaining contribution $\sum_{i=v+1}^{m} \Psi_k(i) = \Omega_k - \sum_{i=1}^{m} D_k(i) - \sum_{i=v+1}^{m} S_k(i)$ is distributed to tenants with index $[v+1, \ldots, m]$ in proportion to their total contributions. Some heuristics can be employed to speed up the index searching. First, searching should start from index $u+1$ because tenants with index $[1,\ldots u]$ are contributors. Second, we can use binary search strategy to find two successive indexes *v, v*+1 still cannot be satisfied. Namely, the following inequality (1) and (2) must be satisfied:

$$S_k(v) + \frac{\sum_{i=v}^{m} \Psi_k(i) \times \Lambda(v)}{\sum_{i=v}^{m} \Lambda(i)} \geq D_k(v) \tag{9.1}$$

$$S_k(v+1) + \frac{\sum_{i=v+1}^{m} \Psi_k(i) \times \Lambda(v+1)}{\sum_{i=v+1}^{m} \Lambda(i)} < D_k(v+1) \tag{9.2}$$

where $\sum_{i=v}^{m} \Psi_k(i)$ and $\sum_{i=v+1}^{m} \Psi_k(i)$ represent the remaining contributions that will be re- distributed to tenants with unsatisfied demands. Once the index *v* is determined, we can calculate the remaining contribution. The tenants with index $[1,\ldots v]$ receive shares capped at their demands (Lines 16-17), and the tenants with index $[v+1, \ldots, m]$ receive theirinitial shares plus the remaining resource in proportion to their contributions (Lines 19-20).

Algorithm 2 Inter-tenant Resource Trading (IRT)

```
Input: $D = \{D(1), ...., D(m)\}, S = \{S(1), ...., S(m)\}, \Omega$
Output: $S' = \{S'(1), ...., S'(m)\}$
Variables: $[i, C(i), \Lambda(i), U(i), V(i), \Psi(i)] \leftarrow 0$
 1: for $resource\_type\ \ k = 1$ to $p$ do
 2:     for $Tenant\ \ i = 1$ to $m$ do
 3:         /*Allocate each tenant $(i)$ her initial share $S(i)$ */
 4:         $S'_k(i) \leftarrow S_k(i)$
 5:         $U_k(i) \leftarrow D_k(i)/S_k(i)$
 6:         if $S_k(i) \geqslant D_k(i)$ then
 7:             $C_k(i) \leftarrow S_k(i) - D_k(i)$
            /*Calculate tenant(i)'s total contribution $\Lambda(i)$ on all type of resource */
 8:             $\Lambda(i) \leftarrow \Lambda(i) + C_k(i)$
 9: for $resource\_type\ \ k = 1$ to $p$ do
10:     Sort $U_k(i)$ in ascending order;
11:     Find the index $u$ so that $U_k(u) < 1 \leqslant U_k(u+1)$
12:     for $Tenant\ \ i = u+1$ to $m$ do
13:         $V_k(i) \leftarrow (D_k(i) - S_k(i))/\Lambda(i)$
14:     Sort $V_k(i)$ in ascending order;
15:     Find the index $v$ using binary search algorithm so that Equation (1) and (2) are satisfied;
16:     for $Tenant\ \ i = 1$ to $v$ do
17:         $S'_k(i) \leftarrow D_k(i)$   /*share is capped by demand*/
        /*Calculate the remaining contributions for re-allocation*/
18:     $\sum_{i=v+1}^{m} \Psi_k(i) \leftarrow \Omega_k - \sum_{i=1}^{v} D_k(i) - \sum_{i=v+1}^{m} S_k(i)$
19:     for $Tenant\ \ i = v+1$ to $m$ do
20:         $S'_k(i) \leftarrow S_k(i) + \frac{\sum_{i=v+1}^{m} \Psi_k(i) \times \Lambda(v+1)}{\sum_{i=v+1}^{m} \Lambda(i)}$
```

9.5.3.2. Intra-tenant Weight Adjustment (IWA)

A tenant usually needs more than one VM to host her applications. Workloads in different VMs may have dynamic and heterogeneous resource requirements. Thus, dynamic resource flows among VMs belonging to the same tenant can prevent loss of tenant's asset. In F2C, we use IWA to adjust the resource among VMs belonging to the same tenant. We allocate *share* (or weight) for each VM using a policy similar to WMMF. For each type of resource, we first reset each VM's current weight to its initial share. However, if the allocation made to a VM is more than its demand, its allocation should be capped at its real demand, and the unused share should be re-allocated to its sibling VMs with unsatisfied demands. In contrast to WMMF that re-allocates the unused resource in proportion to VMs' share values, we re-allocate the excessive resource share to the VMs in the ratio of their unsatisfied demands. Note that, once a VM's resource share is determined, the resource allocation made to the VM is simply determined by the function $share \xrightarrow{f_2} resource$.

Algorithm 3 Inter-tenant Weight Adjustment (IWA)

```
Input: $d = \{d(1), ..., d(n)\}, s = \{s(1), ..., s(n)\}, S$
Output: $s' = \{s'(1), ..., s'(n)\}$
Variables: $[j, \Gamma, \Phi] \leftarrow 0$
/* Allocate initial share $s(j)$ to each VM($j$) */
 1: $\Phi \leftarrow S - \sum_{j=1}^{n} s(j)$   /*Calculate the difference of initial total share and new allocated capacity */
 2: for $VM\ j = 1$ to $n$ do
 3:     if $d(j) \geqslant s(j)$ then
 4:         $\Gamma \leftarrow \Gamma + (d(j) - s(j))$ /*total unsatisfied demand*/
 5:     else
 6:         $\Phi \leftarrow \Phi + (s(j) - d(j))$ /*total remaining capacity*/
    /* distribute remaining capacity to VMs with unsatisfied demand */
 7: for $VM\ j = 1$ to $n$ do
 8:     if $d(j) \geqslant s(j)$ then
 9:         $s'(j) \leftarrow s(j) + \frac{d(j)-s(j)}{\Gamma} \times \Phi$
10:     else
11:         $s'(j) \leftarrow d(j)$
```

Algorithm 3 shows the pseudo-code for IWA. A tenant with n VMs is allocated with total resource share S. Note that S is a global allocation vector which corresponds to the output of Algorithm 2. Thus, Algorithm 3 is performed accompanying with Algorithm 2. For each tenant, we first calculate her total unsatisfied demand and total remaining capacity for re-allocation, respectively (Lines 2 to 6), and then distribute the remaining capacity to unsatisfied VMs in ratio of their unsatisfied demands (Lines 7 to 11). As VM provisioning is constrained to physical hosts' capacity, it is desirable to adjust weights of VMs on the same physical node, rather than across multiple nodes. In practice, we execute the IWA algorithm only on each single node.

9.5.4. Experimental Evaluation

RRF is implemented on top of Xen 4.1. The prototype F2C is deployed in a cluster with 10 nodes. The following workloads with diversifying and variable resource requirements are used to evaluate resource allocation fairness and application performance.

TPC-C: We use a public benchmark DBT-2 as clients, to simulate a complete computing environment where a number of users execute transactions against a database. The clients and MySQL database server run in two VMs separately. We assume these two VM belong to the same tenant. We configure 600 terminal threads and 300 database connections. We evaluate its application performance by throughput (transactions per minute).

RUBBoS (Rice University Bulletin Board System) [61]: it is a typical multi-tier web benchmark. RUBBoS simulates an on-line news forum like slashdot.org. We deploy the benchmark in a 3-tier architecture using Apache 2.2, Tomcat 7.0, and MySQL 5.5.

Kernel-build: We compile Linux 2.6.31 kernel in a single VM. It generates a moderate and balanced load of CPU and memory.

Hadoop: We use Hadoop WordCount micro-benchmark to setup a virtual cluster consisting of 10 worker VMs and one master VM. The 10 worker VMs are evenly distributed in the 10 physical machines and co-run with other three workloads.

On our test bed, we consider multiple tenants sharing the cluster. Each tenant only runs one kind of the above workloads. All workloads are running in VMs whose initial share values are set according to the workloads' average demands. In addition, we can configure the initial share of tenant (i) based on a provisioning coefficient,

$$\alpha = S(i)/\overline{D(I)}$$

which reflects the ratio of initial resource share to the average demand. We continuously launch the tenants' applications to the cluster one by one until no room to accommodate any more applications.

We evaluate F2C by comparing the following alternative approaches for IaaS clouds:

- **T-shirt (static)**: Workloads are running in VMs with static resource provision. It is the current resource model adopted by most IaaS clouds [32].
- **WMMF** (Weighted Max-Min Fairness): WMMF [46] is used to allocate CPU and Memory resources to each VM separately.
- **DRF** (Dominate Resource Fairness): DRF [30] is used to allocate multiple resources to each VM.
- **IWA** (Intra-tenant Weight Adjustment): We conduct only weight adjustment for VMs belonging to the same tenant, without considering inter-tenant resource trading. This is to assess the individual impact of inter-tenant resource trading.
- **RRF (IRT + IWA)**: We conduct hierarchical resource allocation using both inter-tenant resource trading and intra-tenant weight adjustment.

9.5.4.1. Results on Fairness

In general, in a shared computing system with resource contention, every tenant wants to receive more resource or at least the same amount of resource than that she buys. We call it *fair* if a tenant can achieve this goal (i.e., sharing benefit). In contrast, it is also possible the total resource a tenant received is less than that without sharing, which we call *unfair* (i.e., sharing loss). Thus, we define the economic fairness degree $\beta(i)$ for tenant i in a time window T as follows:

$$\beta(i) = \sum_{i=1}^{T} S_t^{'}(i) \,/\, (T \times S(i)).$$

It represents the ratio of average resource share received to the tenant's initial share in the time window T, or more exactly speaking, it denotes the ratio of resource value to tenant's payment. $\beta(i)$=1 implies absolute economic fairness. $\beta(i) > 1$ implies the tenant benefits from resource sharing while $\beta(i) < 1$ implies the tenant loses her asset.

We evaluate the economic fairness of different schemes for the four workloads in a long period of time. All VMs' capacity are provisioned based on their average demands (i.e., $\alpha = 1$). Figure 9.5 shows the comparison of economic fairness of different resource allocation schemes. Each bar shows the average result of the same workload run by multiple tenants. Overall, RRF achieves much better economic fairness than other approaches.

Specifically, RRF leads to smaller difference of β between different applications, indicating 95% economic fairness (geometric mean) for multi-resource sharing among multi-tenants.

We make the following observations.

First, the T-shirt (static) model achieves 100% economic fairness as VMs share nothing with each other. However, it results in the worst application performance for all workloads, as shown in Figure 9.6.

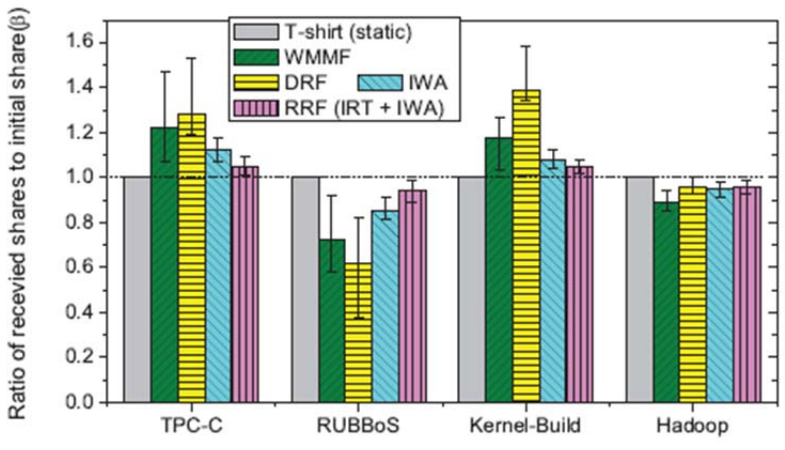


Figure 9.5: Comparison of economic fairness of several resource allocation schemes.

Second, both WMMF and DRF show significant differences on β for different workloads. As all WMMF-based algorithms always try to satisfy the demand of smallest applications first, both kernel-build and TPC-C gain more resources than their initial shares. This effect is more significant for DRF if the application shows tiny skewness of multi-resource demands (such as kernel-build). DRF always completely satisfies the demand of these applications first. Thus, DRF and WMMF are susceptible to application's load patterns. Applications with large skewness of multi-resource demands usually lose their asset. Even for a single resource type, large deviation of resource demand also lead to distinct economic unfairness.

Third, workloads with different resource demand patterns show different behavior in resource sharing. RUBBoS has a cyclical workload pattern, its resource demand shows the largest temporal fluctuations. When the load is below its average demand, it contributes resource to other VMs and thus loses its asset. However, when it shows large value of D_tpiq{Spiq, its demand always can not be fully satisfied if there exists resource contention.

Thus, RUBBoS has smaller value of β than other applications. For TPC-C and Kernel-build, although they show comparable demand deviation and skewness with RUBBoS, they has much more opportunities to benefit from RUBBoS because their absolute demands are much smaller. For Hadoop, although it requires a large amount of resource, it demonstrates only a slight deviation of resource demands, and there is few opportunities to contribute resource to other VMs (except in its *reduce* stage).

Fourth, inter-tenant weight adjustment (IWA) allows tenants properly distribute VMs' spare resource to their sibling VMs in proportional to their unsatisfied demands. It guarantees that tenants can effectively utilize their own resource. Inter-tenant resource trading can further preserve tenants' asset as each tenant tries to maximize the value of spare resource. In addition, RRF is immune to free-riding.

We also studied the impact of different α values. For space limitation, we omit the figures and briefly discuss the results. When we decrease the provisioning coefficient α (i.e., reduce the resource provisioned for applications), the β of all applications approach to one. In contrast, a larger α leads to a decrease of β. That means applications tends to preserve their resource when there exist intensive resource contention. Nevertheless, a larger value of α implies better application performance.

9.5.4.2. Improvement of Application Performance

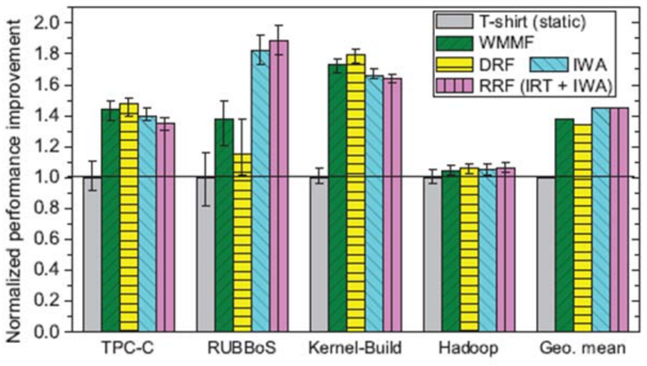


Figure 9.6: Comparison of application performance improvement of several resource allocation schemes.

Figure 9.6 shows the normalized application performance for different resource allocation schemes. All schemes provision resource at applications' average demand ($\alpha = 1$). In the T-shirt model, all applications show the worst performance and we refer it as a baseline. In other models, all applications show performance improvement due to resource sharing. For RUBBoS, RRF leads to much more application performance improvement than other schemes. This is because, RRF provides two mechanisms (IRT + IWA) to preserve tenants' asset, and thus RRF allow RUBBoS reveive more resource than other schemes, as shown in Figure 9.5. For other workloads, RRF is also comparable to the other resource sharing schemes. In summary, RRF achieves 45% performance improvement for all workloads on average (geometric mean). DRF achieves the best performance for Kernel-build and TPC-C, but achieves very bad performance for RUBBoS. It shows the largest performance differentiation for different workloads. DRF always tends to satisfy the demand of the application with smallest dominant share, and thus applications that have resource demand of small sizes or small skewness always benefit more from resource sharing. In contrast, the performance of Hadoop shows slight variations between different allocation schemes due to its rather stable resource demands.

9.6. Related work on Resource Management

In this section, we review the related work of resource management from the aspects of performance, fairness, energy and power cost, as well as monetary cost. For more related work on resource management, we refer readers to three comprehensive surveys [44][47][70].

9.6.1. Resource Utilization Optimization

For resource management, different resource allocation strategies can lead to significantly varied resource utilization [11]. Various resource allocation approaches have therefore been proposed for different workloads and systems/platforms [24][35][62][66]. For MapReduce workloads, Tang et al. [62] observed that different job submission order can have a significant impact on the resource utilization and therefore improved the resource utilization for MapReduce cluster by re-ordering the job submission order of arriving MapReduce jobs. Grandl et al. [35] proposed a dynamic multi-resource packing system called Tetris that improves the resource utilization by packing tasks to machines based on their requirements along multiple resources. Moreover, resource sharing is another approach to improve the resource utilization in a multi-tenant system by allowing overloaded users to possesses unused resources from underloaded users [63]. In addition, task/VM migration and consolidation [15][22][41], widely used in cloud computing, is also an efficient method to improve resource utilization of a single machine.

9.6.2. Power and Energy Cost Saving Optimization

Power and Energy is a big concern in current data centers and supercomputers, which consists of hundreds of servers. Efficient resource management is non-trivial for power and energy cost saving. There are a number of techniques proposed to alleviate it. One intuitive approach is shutting down some low-utilized servers [48][59]. Moreover, virtual machine (VM) migration and consolidation is an effective approach to reduce the number of running machines and in turn save the power and energy cost [21][60]. Finally, we refer readers to a survey [16] for more solutions.

9.6.3. Monetary Cost Optimization

The monetary cost optimization have become a hot topic in recent years, especially for cloud computing. A lot of job scheduling and resource provisioning algorithms have been proposed by leveraging market-based techniques [28], rule-based techniques [53] and model-based approaches [18]. Many relevant cost optimizations can be found in databases [37], Internet [51], distributed systems [36], grid [6], and cloud [28].

9.6.4. Fairness Optimization

Fairness is an important issue in multi-users computing environment. There are various kinds of fair policies in the traditional HPC and grid computing, including round-robin [26], proportional resource sharing [68], weighted fair queuing [25], and max-min fairness [4]. In comparison, max-min fairness is the most popular and widely used policy in many existing parallel and distributed systems such as Hadoop [71], YARN [65], Mesos [40], Choosy [31], Quincy [43]. Hadoop [71] partitions resources into slots and allocates them fairly across pools and jobs. In contrast, YARN [65] divides resources into containers (i.e., a set of various resources like memory, cpu) and tries to guarantee fairness between queues. Mesos [40] enables multiple diverse computing frameworks such as Hadoop, Spark sharing a single system. It proposes a distributed two-level scheduling mechanism called resource offers that decides how many resources to offer each framework and framework decides which resources to accept or which computation to

run on them. Choosy [31] extends the max-min fairness by considering placement constraints. Quincy [43] is a fair scheduler for Dryad that achieves the fair scheduling of multiple jobs by formulating it as a min-cost flow problem. In addition to the single-resource fairness, there are some work focusing on multi-resource fairness, including Dominant Resource Fairness (DRF) [30] and its exten- sions [17][45][58][69].

9.7. Open Problems

Despite many recent efforts on resource management, there are a number of open problems remained to be explored in future. We briefly elaborate them from the following aspects.

9.7.1. SLA Guarantee for Applications

In practice, users' applications are often with different performance (e.g., latency, through-put) and resource requirements. For example, latency-sensitive applications in memcached require high response time whereas batch jobs in Hadoop are often require high through-put [38]. When these applications run together in a shared computing system, it becomes a challenge work to provide the quality of services for each application. Despite many research efforts [20][52] have been made, there is a lack of systematic approach that takes into account different typed resources and application requirements integrally. Most of them either focus on a specific resource type (e.g., network flow) [20][38] or a kind of applications [49]. It remains further studies to systematically ensure SLA guarantee for different applications.

9.7.2. Various Computation Models and Systems

As listed in Section 1.3, there are a number of computation models as well as computing systems proposed for different applications in recent years. From a user's perspective, it becomes a headache and time-consuming problem for the user to choose and learn those computation models and corresponding computing systems. Designing a general computing and resource management system like operation system becomes a trend and challenging issue.

9.7.3. Exploiting Emerging Hardware

A number of emerging hardware have been available at different layers. For example, in the storage layer, we now have Solid State Disk (SSD) and Non-volatile RAM (NVRAM), which are much faster than HD. In the computation layer, there are a set of accelerators such as GPU, APU and FPGA. Moreover, in the network layer, remote direct memory access (RDMA) is an efficient hardware tool for speeding network transfer. For a computing system, it is important to adopt these emerging hardware to improve the performance of applications. Currently, the study on this aspect is still at early stage. More research efforts are required to efficiently utilize these emerging hardware at different layers for existing computing systems.

9.8. Summary

In this chapter, we have discussed the importance of resource management for big data processing and surveyed a number of existing representative large-scale data processing systems. One of the classic issues for resource management is fairness. The chapter reviewed the memoryless fair resource allocation policies for existing systems and showed their unsuitability for cloud computing by presenting three problems. A new Long-Term Resource Allocation (LTRF) policy was then proposed to address these problems, and we provably and experimentally validate the merits of the proposed policy. This chapter next focused on the resource management for virtual machines on the cloud, considering VM migration and consolidation in the cloud environment. Finally, there are many open problems that need more research efforts in this field.

References

[1]. Giraph. In *http://giraph.apache.org/.*

[2]. Hadoop. In *http://hadoop.apache.org/.*

[3]. Loan agreement. In *http://en.wikipedia.org/wiki/Loan_agreement.*

[4]. Max-min fairness (wikipedia). In *http://en.wikipedia.org/wiki/Max-min_fairness*.

[5]. Software-defined networking. In *https://en.wikipedia.org/wiki/Software-defined-networking*.

[6]. Storage resource management. In https://en.wikipedia.org/wiki/Storage Resource Management.

[7]. Storm. In *http://storm-project.net/.*

[8]. Tez. In *https://tez.apache.org/*.

[9]. S. Abrishami, M. Naghibzadeh, and D. H. Epema. Cost-driven scheduling of grid workflows using partial critical paths. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1400–1414, 2012.

[10]. Amazon. *http://aws.amazon.com/solutions/case-studies/.*

[11]. V. Anuradha and D. Sumathi. A survey on resource allocation strategies in cloud computing. In *Information Communication and Embedded Systems (ICICES), 2014 International Conference on*, pages 1–7. IEEE, 2014.

[12]. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T.

Kaftan,

M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[13]. AutoScaling. http://aws.amazon.com/autoscaling.

[14]. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[15]. A. Beloglazov and R. Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 24(7):1366–1379, 2013.

[16]. A. Beloglazov, R. Buyya, Y. C. Lee, A. Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82(2):47-111,2011.

[17]. A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 4:1–4:15, New York, NY, USA, 2013. ACM.

[18]. E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems*, 27(8):1011–1026, 2011.

[19]. L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu

schedulers in xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[20]. M. Chowdhury and I. Stoica. Coflow: An application layer abstraction for cluster networking. In *ACM Hotnets*, 2012.

[21]. C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd con- ference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[22]. A. Corradi, M. Fanelli, and L. Foschini. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32:118–127, 2014.

[23]. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[24]. C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[25]. A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queue- ing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM.

[26]. M. Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[27]. A. Faraz, L. Seyong, T. Mithuna, and V. T. N. Puma: Purdue mapreduce bench- marks suite. Technical Report EECS-2012-Oct, School of Electrical and Computer Engineering, Purdue University, Oct 2012.

[28]. H. M. Fard, R. Prodan, and T. Fahringer. A truthful dynamic workflow scheduling mechanism for commercial multicloud environments. *Parallel and Distributed Sys- tems, IEEE Transactions on*, 24(6):1203–1212, 2013.

[29]. L. George. *HBase: the definitive guide.* " O'Reilly Media, Inc.", 2011.

[30]. A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dom- inant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[31]. A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Confer- ence on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM.

[32]. D. Gmach, J. Rolia, and L. Cherkasova. Selling t-shirts and time shares in the cloud. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 539–546, Washington, DC, USA, 2012. IEEE Computer Society.

[33]. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*, pages 599–613, 2014.

[34]. Google. *https://cloud.google.com/customers/*.

[35]. R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIG- COMM*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.

[36]. J. Gray. Distributed computing economics. *Queue*, 6(3):63–68, 2008.

[37]. J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *arXiv preprint cs/9809005*, 1998.

[38]. M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues dont matter when you can jump them! In *Proc. NSDI*, 2015.

[39]. A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[40]. 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 cen- ter. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Asso- ciation.

[41]. C.-H. Hsu, S.-C. Chen, C.-C. Lee, H.-Y. Chang, K.-C. Lai, K.-C. Li, and C. Rong. Energy-aware task consolidation technique for cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 115–121. IEEE, 2011.

[42]. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data- parallel programs from sequential building blocks. In

Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[43]. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[44]. B. Jennings and R. Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014.

[45]. I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the 2013 International Conference on Au- tonomous Agents and Multi-agent Systems*, AAMAS '13, pages 351–358, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[46]. S. Keshav. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[47]. K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software-Practice and Experience*, 32(2):135–64, 2002.

[48]. W. Lang and J. M. Patel. Energy management for mapreduce clusters.

Proceedings of the VLDB Endowment, 3(1-2):129–139, 2010.

[49]. N. Lim, S. Majumdar, and P. Ashwood-Smith. A constraint programming-based re- source management technique for processing mapreduce jobs with slas on clouds. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 411–421. IEEE, 2014.

[50]. H. Liu and B. He. Reciprocal resource fairness: Towards cooperative multiple- resource fair sharing in iaas clouds. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 970–981, Nov 2014.

[51]. R. T. Ma, D. M. Chiu, J. Lui, V. Misra, and D. Rubenstein. Internet economics: The use of shapley value for isp settlement. *IEEE/ACM Transactions on Networking (TON)*, 18(3):775–787, 2010.

[52]. J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource manage- ment in multi-tenant distributed systems.

[53]. M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 22. IEEE Computer Society Press, 2012.

[54]. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Cza- jkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on*

Management of Data, SIGMOD'10, pages 135–146, New York, NY, USA, 2010. ACM.

[55]. M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 423–430. IEEE, 2012.

[56]. H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: elastic distributed resource scaling for infrastructure-as-a-service. In *USENIX ICAC*, 2013.

[57]. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so- foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[58]. D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Ex- tensions, limitations, and indivisibilities. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 808–825, New York, NY, USA, 2012. ACM.

[59]. K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 111–122. IEEE, 2003.

[60]. P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. In *ACM SIGARCH Computer Architecture*

News, volume 34, pages 66–77. IEEE Computer Society, 2006.

[61]. RUBBOS. *http://jmob.ow2.org/rubbos.html*.

[62]. S. Tang, B.-S. Lee, and B. He. Mrorder: Flexible job ordering optimization for online mapreduce workloads. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 291–304. Springer Berlin Heidelberg, 2013.

[63]. 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 *Proceedings of the 28th ACM In- ternational Conference on Supercomputing*, ICS '14, pages 251–260, New York, NY, USA, 2014. ACM.

[64]. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyck- off, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[65]. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotia- tor. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[66]. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large- scale cluster management at google with borg. In *Proceedings of*

the Tenth European Conference on Computer Systems, page 18. ACM, 2015.

[67]. C. A. Waldspurger. Memory resource management in vmware esx server. In Pro- ceedings of the 5th Symposium on Operating Systems Design and implementation- Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading, OSDI '02, pages 181–194, New York, NY, USA, 2002. ACM.

[68]. C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[69]. W. Wang, B. Li, and B. Liang. Dominant resource fairness in cloud computing sys- tems with heterogeneous servers. In INFOCOM, 2014 Proceedings IEEE, pages 583–591, April 2014.

[70]. R. Weingrtner, G. B. Brscher, and C. B. Westphall. Cloud resource management: A survey on forecasting and profiling models. Journal of Network and Computer Applications, 47:99 – 106, 2015.

[71]. T. White. Hadoop: The Definitive Guide. O'Reilly, first edition edition, june 2009.

[72]. D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In Proceedings of the 7th ACM SIG- PLAN/SIGOPS International Conference on Virtual Execution Environments, VEE’11, pages 205–216, New York, NY, USA, 2011. ACM.

[73]. R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of data, pages 13–24. ACM, 2013.

[74]. A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In Job Scheduling Strategies for Parallel Processing, pages 44–60. Springer, 2003.

[75]. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[76]. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[77]. J. Zhong and B. He. Medusa: Simplified graph processing on gpus. Parallel and Distributed Systems, IEEE Transactions on, 25(6):1543–1552, 2014.