7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015), Vancouver, BC, Canada, Nov 30-Dec 3, 2015, (c) IEEE.

Providing Fairer Resource Allocation for Multi-tenant Cloud-based Systems

Jia Ru, John Grundy, Yun Yang
School of Software and Electrical Engineering
Swinburne University of Technology
Melbourne, Australia 3122
Email:{rjia, jgrundy, yyang}@swin.edu.au

Jacky Keung
Department of Computer Science
City University of Hong Kong
Hong Kong SAR
Email:Jacky.Keung@cityu.edu.hk

Li Hao
Research & Development Department
Winhong Information Technology Co.,Ltd
Guangzhou, Guangdong, China 510642
Email:liucoolhao@gmail.com

***Abstract*—**
A fundamental premise in cloud computing is trying to provide a more sophisticated computing resource sharing capability. In order to provide better allocation, the Dominant Resource Fairness (DRF) approach has been developed to address the "fair resource allocation problem" at the application layer for multi-tenant cloud applications. Nevertheless conventional DRF only considers the interplay of CPU and memory, which may result in over allocation of resources to one tenant's application to the detriment of others. In this paper, we propose an improved DRF algorithm with 3-dimensional demand vector <CPU, memory, vdisk> to support disk resources as the third dominant shared resource, enhancing fairer resource sharing. Our technique is integrated with LINUX '*Cgroup*' controls resource utilisation and realises data isolation to avoid undesirable interactions between co-located tasks. Our method ensures all tenants receive system resources fairly, which improves overall utilisation and throughput as well as reducing traffic in an over-crowded system. We evaluate the performance of different types of workload using different algorithms and compare ours to the default algorithm. Results show an increase of 15% resource utilisation and a reduction of 59% completion time on average, indicating that our DRF algorithm provides a better, smoother, fairer high-performance resource allocation scheme for both continuous workloads and batch jobs.

Keywords–cloud computing, scheduling, algorithm, DRF, multi-tenancy

I. Introduction

Cloud computing enables resource sharing but different tenants may have diverse requirements for each resource (CPU, memory and I/O) for their hosted applications. Cloud providers are strongly driven by consumer needs, and market-oriented resource management strategies are becoming necessary to regulate strong demand for resources, this is to achieve optimum resource utilisation and providing to their clients/tenants with acceptable performance [1]. But different kinds of cloud resources exhibit different levels of dynamic and interactive behaviours according to user demands. Large clusters running parallel processing frameworks such as MapReduce [2] are becoming commonly used techniques. Different applications could be classified as communication-intensive, CPU-intensive and data-intensive and thus the requirements of different tenants become more diverse [3]. There are studies on dynamic resource allocation and performance isolation between applications and tenants in a shared computing environment and becoming a new challenge [4]. Both over/under-allocation of resources are undesirable and will adversely impact the tenant itself and others in the shared cloud environment.

The Max-min fairness algorithm is one of the most popular resource allocation mechanisms being used, originally proposed for computer networks to facilitate better scheduling capability. Nowadays, it is being widely used in cloud computing systems, with different implementations such as Choosy [3] and DRF [5]. These are attempts to optimize the minimum resource allocation needed and received by each prospective tenant [5]. Moreover, the weighted max-min fairness model focuses on data resource isolation, in that each tenant is assured of receiving their needed share without considering requirements of other tenants [5]. However, most of fairness resource allocation research to date has been on homogenous environments. Today's computing environments are mostly heterogeneous and thus the allocation policies implemented by a single resource abstraction will not necessarily satisfy the needs of different cloud applications.

Dominant Resource Fairness or DRF for short [5] is a method to provide fair allocation for heterogeneous resources. DRF attempts to maximise the "minimum dominant share" of resources for all users. DRF attempts to fairly distribute memory and CPU resources among these different types of jobs in a mixed-workload cluster [6]. But it does not consider disk I/O resources and network bandwidth, which is its main weakness. Tasks consume different resources simultaneously, and tasks may be bottlenecked and blocked on these different resources. Without controlling the disk I/O, some tasks will exhaust disk I/O, which will incur disk blocking of other tasks and increase their completion time. Improved job completion time after implementing a disk optimization approach represents a best-case scenario [7].

In this paper we present an enhancement to DRF to support fair, efficient allocation of cloud resources to each cloud tenant. First, our algorithm considers not only CPU and memory resources, but also disk resources as the third candidate to avoid over allocation. If not considering disk I/O, it is rather easy to over allocate disk resources for the tasks that need a relatively smaller share of CPU and memory. Second, co-located tasks might compete for I/O resources and interfere with each other. Isolating disk I/O resources is essential to deliver predictable performance. Our algorithm uses '*Cgroup*' to limit and control each job's resource usage and I/O, aiming to realise both logical resource isolation and physical resource isolation. For example, according to job requirements, tasks are configured for different I/O, then assigned to different physical disk partitions without any interaction among the tenants. This enables different tasks from multiple tenants to share resources reasonably and fairly, improving overall system resource utilisation and throughput. In reality many tenant applications do not necessarily need a large amount of resources allocated, and they are not concerned with how much resource is being allocated to them. Our method is able

to determine the dominant share level, even if the smallest required resource level can be defined as the dominant share. Our proposed technique provides a comprehensive handling of multiple resource types required for multi-tenant cloud platforms, a significant improvement over the conventional naïve DRF algorithm based on results shown in this study.

Section II discusses related work and Section III describes the motivation for our work by an experimental example. Section IV presents our proposed modifications to the conventional DRF algorithm. Section V presents observations and results, and Section VI concludes the work and outlines future work.

II. Related work

A. Max-min fairness resource allocation

Max-min fairness is a flexible resource allocation mechanism used in datacenter schedulers, networks, operating systems and queueing systems [3] [8]. In heterogeneous resource environments, an increase in the allocation to any participant necessarily results in a decrease in the allocation to some other participants with an equal or smaller allocation [9]. Compared to a first-come first-served (FCFS) scheduling policy, max-min fairness provides traffic shaping to avoid resource congestion.

B. Dominant resource fairness (DRF)

DRF uses the concept of a "dominant resource" to compare multi-dimensional resources. In a multi-resource environment, resource allocation should be determined by the dominant share of an entity (user or queue), which is the maximum share that the entity has been allocated of any resource (memory or CPU). In a nutshell, DRF tries to maximise the minimum dominant share across all tasks using shared resources [6]. For instance, when user a runs CPU-heavy jobs (e.g. Storm-on-YARN [10]) and user b runs memory-heavy jobs (e.g. MapReduce [2]), DRF seeks to equalise the CPU share of a with the memory share of b. Eventually, DRF will allocate more CPU and less memory to a, and allocate less CPU and more memory to b. In a homogeneous resource environment, all the tasks require the same type of resources and hence DRF reduces to max-min fairness for the resource [6] [5]. DRF algorithm [5] identifies some significant allocation properties, such as sharing incentive, strategy-proofness, envy-freeness and more importantly Pareto efficiency. The strength of DRF lies in these properties which are satisfied, especially when these properties are trivially satisfied by max-min fairness for a single resource, but are important in multiple resources. DRF provides incentives for users to share resources by guaranteeing that no user is better off in a system in which resources are statically and equally partitioned among users. DRF ensures that users do not gain a better allocation by lying about their resource demands. Moreover, DRF allocates all available resources subject to satisfying the other properties, and without preempting existing allocations. DRF ensures that no user prefers the allocation of another user. [5]

The Capacity Scheduler is designed to run Hadoop applications as a shared, multi-tenant cluster friendly to maximise the throughput and utilisation of the cluster [11]. Capacity Scheduler has two Resource Calculator implementations. A default resource calculator is where resources are allocated based on memory alone. The other is a Dominant Resource Calculator based on the DRF model of resource allocation only with consideration of CPU and memory [6]. DRF can also work on fair scheduler, where it is set as a fairness policy at each level of the scheduling hierarchy [12]. Its dominant share (current usage of resources) is defined as the maximum ratio between a resource type in the current usage and a resource type in the cluster. However, DRF on fair scheduler does not exactly know what the multi-resource analog to this should be and hardly assigns each scheduled queue a fair share of each resource [12].

C. Cgroup - Control groups

Cgroup, an acronym for 'control groups', is a Linux kernel proposed by Google, that limits, accounts for and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes [13]. In '*Cgroup*', a task is referred to as an OS process. '*Cgroup*' associates tasks with different parameters for one or more subsystems that use task grouping facilities to treat groups of tasks in particular ways. The task groups can be set to not exceed configured CPU, memory and disk I/O boundaries to limit and isolate resources, but some groups with higher priority can receive larger share of CPU utilisation and disk I/O throughput [14]. '*Cgroup's*' subsystem 'blkio' implements the block I/O controller [15]. Internally, CFS (completely fair scheduler) is a new 'desktop' process scheduler [16] to maintain fairness in providing processor time to tasks which is used in our allocation policy. Each process should gain a fair amount of the processor. When one or more tasks are not received a fair amount of time relative to others, those unfair tasks should be given time to execute [17]. The amount of time provided is measured by virtual time, the smaller time a task is permitted access to the processor [17]. '*Cgroup*' realises resource isolation and I/O control. The tasks with different types or of diverse tenants are set with specific I/O by '*Cgroup*' and assigned to physical disk partitions with corresponding I/O without any interaction.

D. YARN - the next generation Hadoop

YARN [10], as the next-generation Hadoop [18], computing platform separates its functions into 2 layers: platform layer for resource management and framework layer coordinating application execution [19]. It decouples the programming model from resource management infrastructure, orchestrates various running applications with different resource requirements and arbitrates all kinds of resources to tenants. YARN's scheduling component is more granular and dynamic, not strictly partitioning nodes. However, currently it is only sensitive to memory and CPU [10].

III. Motivating Example

Assume a cloud system with 24 virtual CPUs (vCPUs), 36 GB RAM, 54 virtual disks (vdisks), a resource sharing degree of $Lev = 1$, and 3 users (tenants). Our algorithm introduces a "resource sharing degree" concept, in which the lower the degree, the more dominant the resource for a task. User a runs tasks with resource requirement 3-dimensional demand vector <2 vCPU, 4 GB, 3 vdisk>, User b runs tasks with requirement <3 vCPU, 2 GB, 6 vdisk>, and User c runs tasks with requirement <1 vCPU, 3 GB, 6 vdisk>.

In our proposed DRF allocation algorithm, CPU, memory and disk resources are all considered. The restriction of I/O speed and amount of shared storage I/O can be set for each task, job, tenant or group of tenants. $Lev = 1$ indicates the

most dominant resource value. Therefore, after calculating the fraction of required resources to the total resource, the largest one is selected as the dominant share. User a requires $\{\frac{1}{12}$ CPU, $\frac{1}{9}$ memory, $\frac{1}{18}$ vdisk$\}$, and its dominant share is memory. User b needs $\{\frac{1}{8}$ CPU, $\frac{1}{18}$ memory, $\frac{1}{9}$ vdisk$\}$, and its dominant share is CPU. User c requires $\{\frac{1}{24}$ CPU, $\frac{1}{12}$ memory, $\frac{1}{9}$ vdisk$\}$, and its dominant share is disk.

This allocation can be calculated and simplified mathematically as follows. Given x, y and z respectively stand for the number of tasks allocated by proposed DRF to user a, b and c. User a is allocated $<2x$ vCPU, $4x$ GB, $3x$ vdisk $>$, User b is allocated $<3y$ vCPU, $2y$ GB, $6y$ vdisk $>$, and User c is allocated $<z$ vCPU, $3z$ GB, $6z$ vdisk $>$. The total amount of resources assigned to these 3 users are $(2x + 3y + z)$ CPUs, $(4x + 2y + 3z)$ GB, and $(3x + 6y + 6z)$ vdisks.

Our proposed enhanced DRF algorithm attempts to equalise the dominant share of users a, b, and c: $\frac{4x}{36} = \frac{3y}{24} = \frac{6z}{54}$. We should point out that DRF does not always equalise users' dominant share, since as one user's resource requirement is satisfied, the extra resources will be split to other users. If one type of resource is exhausted, the users that do not need that type of resource will still continue receiving higher shares of other types of resources [5].

Equation 1 provides an answer to this problem:

$$\begin{array}{l} \max(x, y, z) \qquad (maximize\ allocations) \\ \text{constraint to} \\ \begin{cases} 2x + 3y + z \leq 24 \quad (CPU\ constraint) \\ 4x + 2y + 3z \leq 36 \quad (memory\ constraint) \\ 3x + 6y + 6z \leq 54 \quad (disk\ constraint) \\ \frac{4x}{36} = \frac{3y}{24} = \frac{6z}{54} \quad (equalise\ dominant\ share) \end{cases} \end{array} \tag{1}$$

Solving this equation, we get x = 4, y = 3, and z = 4 (note that one task must be processed as an entity, so the values of x, y and z must be an integer). Consequentially, user a receives $<$8 vCPU, 16 GB, 12 vdisk$>$, user b receives $<$9 vCPU, 6 GB, 18 vdisk$>$, and user c receives $<$4 vCPU, 12 GB, 24 vdisk$>$. Figure 1 outlines how this could be achieved. User a receives 44.44% of the memory resource; user b receives 37.50% CPU resource; and user c receives 44.44% of the disk resource. The system resource utilisation is considered as very high $<$87.50% CPU, 94.44% memory, 100.00% disk$>$.

In this scenario, the original DRF algorithm does not consider the disk I/O utilisation. User a's dominant share is memory ($\frac{1}{9}$), b's share is CPU ($\frac{1}{8}$), and c share is memory ($\frac{1}{12}$). Based on the Max-min principle, c's task is being executed first, and then for a's task. b's dominant share ratio is the largest and runs next. After several iterations, all the disk I/O resource is exhausted. Finally, a b and c have 3, 3, 4 tasks executed, respectively. System utilization is $<$79% CPU, 83% memory, 100% disk$>$, which is 10% lower than our modified DRF method. Use of the original DRF algorithm easily results in I/O exhaustion, which unfortunately, blocks other resource usage. Our method solves this issue and can also control each kind of resource's usage to avoid some tenants occupying too many resources, which lead to reducing others' performance.

IV. Our approach

A. Proposed DRF algorithm

To address the said issues highlighted, our new DRF allocation algorithm is added to the YARN capacity scheduler to consider CPU, memory, disk resources, as shown in Figure 2. The Queue from YARN's scheduling unit is a logical collection of applications submitted by diverse tenants and can also be regarded as a logical view of the resources on physical nodes. The capacity of each queue specifies the percentage of cluster resources that are available for applications submitted to the queue. Tenants use Yarn to orchestrate applications with differing resource requirements and to arbitrate resources of all kinds. Then the capacity scheduler is enabled by using the Dominant Resource Calculator based on our improved DRF model for resource allocation, where our algorithm is invoked.

Capacity scheduling represents one aspect of YARN resource management capabilities that includes '*Cgroup*', node labels, archival storage, and memory as storage. '*Cgroup*' should be used with capacity scheduling to constrain and manage CPU processes and 'blkio' configures different resource provision for jobs based on diverse tenants' requests. Containers with different resource configurations grant rights to corresponding tasks to use a specific amount of resources and process them. Essentially, a container is a logical bundle of resources bound to a particular cluster node. '*Cgroup*' monitors the running status and allocated resources for each user. It dynamically controls and tunes I/O and other resource allocations to isolate data. With '*Cgroup*' strict enforcement turned on, each task gets only the resources it asks for. Without '*Cgroup*' turned on, the DRF scheduler will do its best to balance allocations out, but unpredictable behaviour may occur. Our method can force some allocation to specified disks via I/O matching. For example, high I/O tasks will be assigned to a disk partition with high I/O capability [6].

Algorithm 1 shows the pseudo code of our modified DRF algorithm. A task is submitted with different resource demands, which is depicted by a 3-dimensional demand vector. In naïve DRF algorithm, only CPU and memory are the elements in the demand vector, for realisation of dominant share. However, we add virtual disk to this resource demand vector. Dos_m is used to denote the demand vector of the next task that user m wants to launch. At each iteration, the scheduler selects the task with the lowest dominant share ready to tun. If that user's next ready task requirement (Dem_m) can be satisfied, then the task will be executed. Next, the scheduler updates the user's resource utilisation and adds the requirement of last running task (Dem_m) to user m's total allocated resources (All_m). When some tasks are finished, the users release their corresponding resources and recalculate the users' total allocated resources. Our algorithm uses $Lev = i$ is $(i = 1, 2, 3)$ to determine the degree of resource sharing: high, medium, and low. When choosing $Lev = 3$, the smallest amount of needed resource for a user is set as its dominant share, which realises the lowest dominant resource share.

During each iteration, the user task with lowest dominant share Dem_m is ready to run. If the equation of *Line 7* is satisfied, the task will be executed and then resource usage is updated for All_m. When tasks finish, related allocated resources are released. Our method can determine different degrees of dominant share. If $Lev = 3$, the smallest amount of needed resource for a user is set as its dominant share, which realises the lowest dominant resource share. Our method uses a binary heap to store each user's dominant share and uses array sort to store and determine the degree of dominant share. Each scheduling decision takes $O(n \log n)$ time for n users.

Schedule	User a		User b		User c		CPU total allocation	Memory total allocation	Disk total allocation
	resource shares	dominant share	resource shares	dominant share	resource shares	dominant share			
User A	<2/24,4/36,3/54>	4/36	<0,0,0>	0	<0,0,0>	***0***	2/24	4/36	3/54
User C	<2/24,4/36,3/54>	4/36	<0,0,0>	***0***	<1/24,3/36,6/54>	6/54	3/24	7/36	9/54
User B	<2/24,4/36,3/54>	***4/36***	<3/24,2/36,6/54>	3/24	<1/24,3/36,6/54>	6/54	6/24	9/36	15/54
User A	<4/24,8/36,6/54>	8/36	<3/24,2/36,6/54>	3/24	<1/24,3/36,6/54>	***6/54***	8/24	13/36	18/54
User C	<4/24,8/36,6/54>	8/36	<3/24,2/36,6/54>	***3/24***	<2/24,6/36,12/54>	12/54	9/24	16/36	24/54
User B	<4/24,8/36,6/54>	***8/36***	<6/24,4/36,12/54>	6/24	<2/24,6/36,12/54>	12/54	12/24	18/36	30/54
User A	<6/24,12/36,9/54>	12/36	<6/24,4/36,12/54>	6/24	<2/24,6/36,12/54>	***12/54***	14/24	22/36	33/54
User C	<6/24,12/36,9/54>	12/36	<6/24,4/36,12/54>	***6/24***	<3/24,9/36,18/54>	18/54	15/24	25/36	39/54
User B	<6/24,12/36,9/54>	***12/36***	<9/24,6/36,18/54>	9/24	<3/24,9/36,18/54>	18/54	18/24	27/36	45/54
User A	<8/24,16/36,12/54>	16/36	<9/24,6/36,18/54>	9/24	<3/24,9/36,18/54>	***18/54***	20/24	31/36	48/54
User C	<8/24,16/36,12/54>	16/36	<9/24,6/36,18/54>	9/24	<4/24,12/36,24/54>	24/54	21/24	34/36	54/54

Figure 1. Resource allocation example using our approach: Each row represents our DRF to making a scheduling decision. Each row shows the share of each user for each kind of resource, the user's dominant share, and the fraction of each resource allocated so far. Our DRF repeatedly selects the user with the lowest dominant share (indicated in bold and red) to launch a task, until no more tasks can be allocated. In the beginning, User a and User c's dominant share fraction are smallest and the same (1/9), and User a is chosen first. Next iteration, User c's dominant share fraction is the smallest, so User c is selected. In the third iteration, User b's dominant share fraction is 0 so far and that only User b has not been given resources, so b is selected. We repeat the iteration until the cluster is saturated. Even if at the first iteration, where User c is being selected, and we could also get the same results.

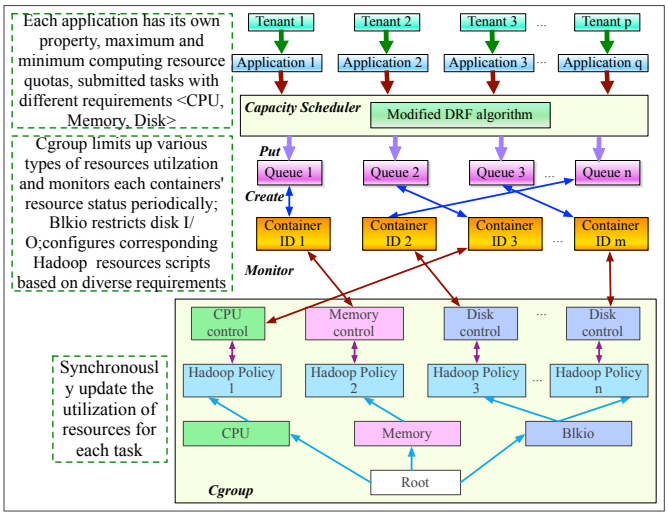

Figure 2. Our proposed Dominant Resource Fairness allocation algorithm

Algorithm 1 Our new DRF algorithm

1: Res = $(r_1, r_2, ..., r_p)$ → total resources capacities
2: Com = $(c_1, c_2, ..., c_p)$ → consumed resources, initial value = 0
3: $\text{Dos}_m (m = 1, 2, ...q)$ → user m 's dominant shares, initial value = 0
4: $\text{All}_m = (a_{m,1}, a_{m,2}, ..., a_{m,p}) (m = 1, 2, ..., q)$ → the resources allocated to user m, initial value = 0
5: Lev = i (i = 1,2,3) → receive the resource amount according the level. Lower the level is lower, the more dominant resource. Level = 1 indicates the most dominant resource value.
6: **Select** user m with lowest dominant share Dos_m
7: Dem_m → demand of user m's next task
8: **if** Com + $\text{Dem}_m \leq Res$ **then**
9: Com = Com +Dem_m → update consumed resources
10: $\text{All}_m = All_m + Dem_m$ → update user m's resource allocation
11: Dos[] = $\text{sort}_{n=1}^{p}(a_{m,n}/r_n)$
12: $\text{Dos}_m = Dos[Dos.length - Lev]$ → determine dominant share degree
13: **else**
return → the cloud cluster is full
14: **end if**

B. Weighted DRF

In practice, allocating resources should not always be equalised between users. We should allocate more resources to the users with priority and running more important, time-sensitive jobs. To achieve this goal, a weighted DRF is derived from our algorithm. In the proposed weighted DRF, each user m has a weight vector $W_m = (w_{m,1}, w_{m,2}, ..., w_{m,p})$, where $w_{m,p}$ means the weight of user m of resource p. When the weights of user m's are being all equal, $w_{m,p} = w_m$, the principle of user m's dominant share changes to $Dos[\,] = sort_{n=1}^{p}(a_{m,n}/(r_n * w_{m,n}))$ (*line12*), which ensures the resource share between diverse users is not equalised.

V. Experiments

To evaluate our algorithm's performance, we employ 4 metrics: resource utilisation, resource restriction effort, completion time and data isolation to evaluate different types of MapReduce-based workloads. Our experiments compare our new algorithm's performance and resource utilisation for both single type of workload and mixed workloads to the naïve DRF. Our experiments contain 2 main parts: exemplar benchmark applications with a single type of workload, and real application with mixed workloads. The latter validates our method's performance for practical applications. Our cloud cluster contains 5 machines each with 16GB of RAM, 2.9 GHz 8 cores Intel Xeon Processors, 3 1TB disk drives with multiple partitions and different I/O speed, running Hadoop 2.6.0 on an Ubuntu server.

A. Exemplar benchmark applications

We selected four of Hadoop's classical benchmarks as follows: 1) **Pi estimator** is a pure ***CPU-intensive*** application that employs a Monte Carlo method to estimate the value of pi. It is nearly "embarrassingly parallel": map tasks are all independent and a single reduce task gathers very little data from map tasks. 2) **Malloc** is a classical ***memory-intensive*** task to allocate unused space for an object whose size in bytes is specified by size and whose value is unspecified. 3) **Read/Write file** is a simple ***I/O-intensive*** task that reads and writes files repeatedly and continuously. Reading frequency equals to writing frequency. 4) **TeraSort** samples the input data and uses map/reduce to sort the data into a total order, which is implemented as a standard MapReduce sort with a custom partitioner that uses a sorted list of $(n-1)$ sampled keys that define the key reduce.

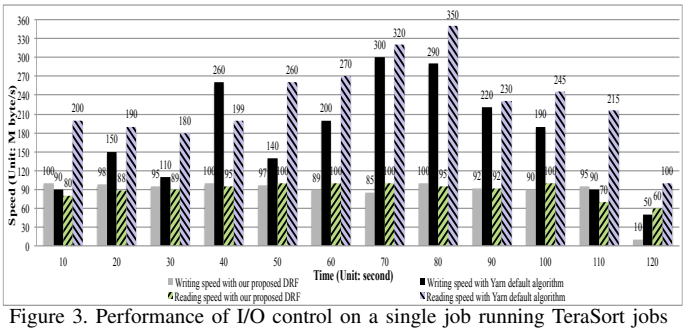

Figure 3. Performance of I/O control on a single job running TeraSort jobs

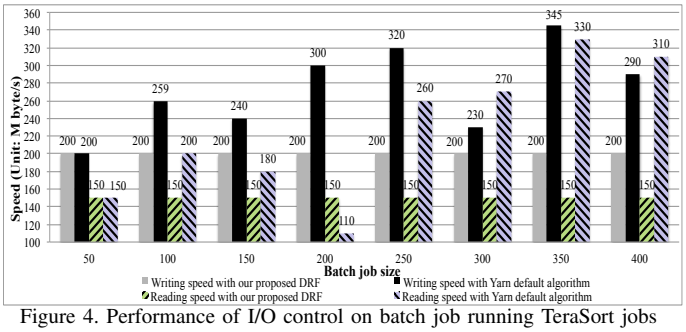

Figure 4. Performance of I/O control on batch job running TeraSort jobs

B. Exemplar benchmark application results

1) Resource utilisation restrictions and data isolation when running TeraSort jobs: We ran TeraSort jobs to test these metics. Figures 3 and 4 present I/O usage restriction effort. Our method can restrict I/O usage flexibly, but the Yarn's default algorithm cannot change I/O arbitrarily. Without I/O control, I/O speed is very discrete and unpredictable. We change each job's I/O and restrict the I/O to a fixed value (100 Mbytes/s in this experiment). However a single job's I/O slightly fluctuates, not exactly 100. For the batch jobs, the entire I/O is set as 200, and the I/O curve is flat, without any change. Our method is more suitable for batch or group jobs. Our method can efficiently control one job or a group of job's I/O to avoid excessive resource preemption. This enables other tenants' jobs, especially small jobs, to have the same right to gain disk I/O, otherwise these jobs would be delayed. It also prevents I/O-heavy tasks from exhausting all the I/O resources. Tasks with different types or from diverse tenants can be manually assigned to different disk partitions without interacting with others. Controlling and restricting I/O is an efficient way to manage disk utilisation and isolate data.

Figure 5 describes the performance of CPU resource usage restriction. After 200 seconds, using the default DRF algorithm, the batch jobs are finished so CPU utilisation is 0. The average CPU usage of the default algorithm is 76.11%. However in our method, CPU utilisation is limited to 50% manually and it is not allowed to allocate all CPU resources to one group of jobs. Thus our method uses much more time to execute jobs but it enables other jobs form different tenants to better share the remaining available resources. After 250 seconds, the job will be finished soon and thus it cannot consume 50% CPU resource. Compared to the naïve algorithm, for only one job, we save 31.82% CPU resource consumption, which can be allocated to other jobs and hence more jobs will be executed concurrently.

2) Resource utilisation of continuous mixed workload (Pi estimator, Malloc and Read/Write file): Figure 6 shows resource utilisation with a continuous mixed workload and

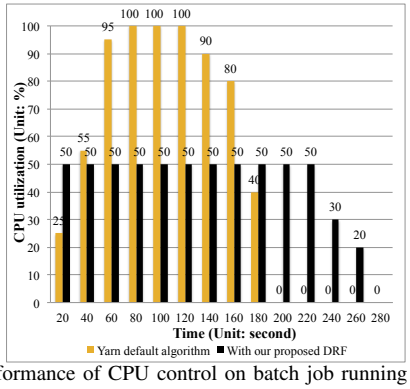

Figure 5. Performance of CPU control on batch job running TeraSort jobs

the cluster is nearly saturated. Our method's CPU utilisation reaches 97.38% on average vs the naïve DRF at only 85.88%, 12% lower on average. Our resource allocation is more steady and fairer to tenants since its CPU fluctuation is only 7%. The average memory utilisation using our algorithm is 91.63%, 7.75 % higher than the default DRF algorithm. At 600 seconds and 2100 seconds, memory utilisation suddenly reduces to 77% and 75%, respectively. Although our method's memory curve is slightly unpredictable, it is still better and flatter than the default algorithm. Disk I/O utilisation with our DRF is 92.625% on average, but the default algorithm's is only 65.625%, and sometimes even only at 20%, and yet our algorithm's utilisation rate can reach 96%. Using our DRF, the disk I/O utilisation improves by nearly 30%.

As shown in Figure 6, a common phenomenon is that the resource utilisation of the default algorithm is not steady, lowly utilised and has apparent variance between peaks and troughs whatever CPU, memory or I/O is. There are some reasons for this, such as when a task reads data from disk and consumes almost all disk I/O, even if there is still some CPU and memory resources available, due to I/O blocking restrictions other tasks cannot read/write data and thus cannot be processed. This results in wasting CPU and memory resource and lowing the utilisation rate. CPU conflict will also influence resource usage. If a group of high CPU-intensive tasks are being executed, which preempt most CPU resources. Even if memory-heavy or I/O-heavy tasks are submitted, they cannot be processed only using memory and I/O without consuming at least some CPU resources. No kind of task only consumes a single resource. The default YARN capacity scheduler only considers memory, but when the memory resource is saturated, other kinds of resource may not reach their full utilisation. In summary, unreasonable resource allocation and coordination between CPU, memory and disk I/O produces this phenomenon. If any one type of resource is not considered or is allocated irrationally (over-used or under-used), it will degrade the entire system's performance. That is also why our method handles CPU, memory and disk together in a fair manner.

3) Completion time and resource utilisation of mixed batch jobs (Pi estimator, Malloc and Read/Write file): Figure 7 shows the resource utilisation of mixed batch jobs. Using our algorithm, CPU utilisation can achieve 100% at sizes of 50, 100, 300, 550, and average utilisation rate reaches 98.42%. Memory utilisation of these size of jobs can reach at least 90%. The memory utilisation without DRF is 82.5% on average, 10 % lower. When the job size increases, the memory utilisation of our DRF is significantly higher. Only when the job size

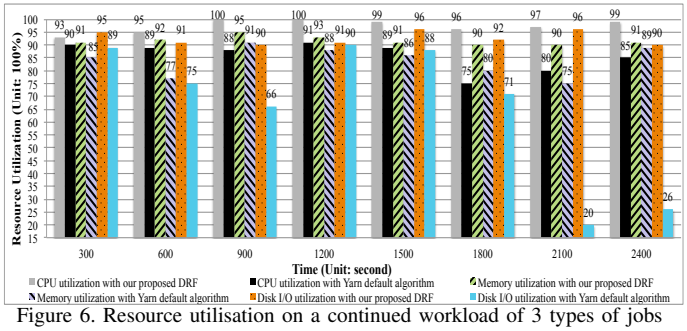

Figure 6. Resource utilisation on a continued workload of 3 types of jobs

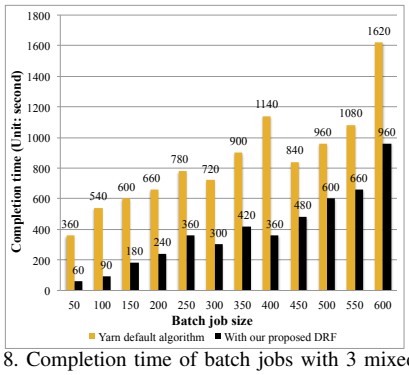

Figure 8. Completion time of batch jobs with 3 mixed types

is 50, the memory utilisation is exceptional. Compared with our benchmark algorithm, memory utilisation is not low. But compared to the other 2 kinds of resources, it is slightly lower. In our method, when one kind of resource is fully utilised, no more tasks will be accepted to execute until exhausted resource is released and the cluster is not saturated. Consequently, when CPU resources are 100% allocated, no more tasks will be accepted, but memory and disk I/O do not get maximum usage.

Our algorithm attempts to maximise all kinds of resource usage. That is why memory and disk I/O utilisations are not low. The default algorithm's CPU utilisation is rather random and low, about 85% and sometimes just 7%. Our algorithm's I/O utilisation is 94.91% on average, but the utilisation without our DRF is only 79.41%. It clearly shows our algorithm can be highly effective in improving disk I/O utilisation. When job size is 350, the default DRF's I/O usage is just 60%, and default algorithm's resource utilisation is slightly lower and will increase the entire system's makespan. Figure 8 shows completion time of different sizes of batch jobs. Completion time of our algorithm is apparently smaller than that without our DRF. The difference between our algorithm and the default one is 457.5 seconds on average. When the size is 600, the difference is largest as 660 seconds. When job size increases, the difference trend is much more significantly. This verifies our algorithm can improve system throughout greatly, especially for large cluster computation.

C. Real applications

Case 1: Enterprise cloud application - telecommunication data analysis: Telecom providers need to find out when and which users prefer calls, data usage, international calls, etc. Providers then push different plans to customers. This data analysis application mines and analyses information from customer usage history, based on classifying and ordering user groups, comparison of different customer behaviour, classification of plans and other techniques. This is a ***CPU-memory-I/O-intensive mixed type application***. A 40G data file containing different clients' data and information as input is spilt into blocks, deployed on different nodes and read into disk and memory, so these operations consume disk I/O and memory. Then the application scans data, and gains the index, filed values and useful information. The complexity of classification is $O(n)$ and classification and analysis process consumes CPU. The recommended plans are stored in multi-dimensional tables, which consumes memory.

Case 2: Number plate image recognition: This License Plate Recognition System (LPRS) recognises a vehicle plate license from images. Edge detection is used to identify points in the digital images with discontinuities. Edge detection calculates every pixel of an image, with complexity $O(n^2)$. A 40G image data file is loaded and read once from disk. Therefore, this application is ***CPU-intensive***.

Case 3: Hadoop log file text search: An enterprise cloud records huge volumes of log files everyday. This application tracks Hadoop's logs to search for error information using a simple lambda expression based on the "error" string, aiming to help us know cluster's health status and weakness. Its complexity is very low and it only consumes little CPU resource. A 40G log file is buffered in memory, read and searched, which consumes little I/O and much more memory resource. Hence, this application is ***Memory-intensive***.

Case 4: Hadoop data migration: In a Hadoop cluster, the input file is split into one or more blocks stored in a set of DataNodes (running on commodity machines). When data volume is huge, tasks split from jobs are deployed on one node, however the needed data may be stored on different nodes even different racks. Thus the system needs to copy other nodes' data to this destination node, which consumes amount of disk I/O and increases data transmission. The 40G telecommunication data file is copied and transmitted among nodes. The data migration application only copies data between different nodes, and is ***I/O-intensive***.

D. Real application results

1) Use case 1: This application is of a mixed type, so all 3 kinds of resource utilisation are relatively high, as shown in Figure 9. Using our DRF algorithm, CPU usage rate reaches 95.33%, memory utilisation is 91.66% and I/O utilisation gets 92.00% on average. During the period of 1200 to 2100 seconds, CPU is 100% fully used and memory utilisation also gets the maximum value - 95.00% but I/O usage looks relatively steady (92.00%). There is a warm-up process that after running an application for a while, the best performance (peak values) appears. That is why our scheduler gets better performance after 900 seconds. However, the default algorithm's resource utilisation is slightly lower. Its CPU usage rate only gets to 79.41% on average, which is 16% lower than our DRF; memory and I/O utilisation is 85.00% and 81.64% respectively, which are both 10% lower than our proposed DRF. Our method efficiently control I/O usage, and the I/O trend is flatter and more steady with only 8% variance (minimum is 88%, maximum is 96%). In comparison, the default algorithm's trend is very unpredictable with 26% in difference.

2) Use case 2: Since this application is CPU-intensive, CPU utilisation is higher than the other 2 kinds of resource in both our DRF (95.33%) and default capacity scheduler (78.67%), sometimes achieving 100% as shown in Figure 10.

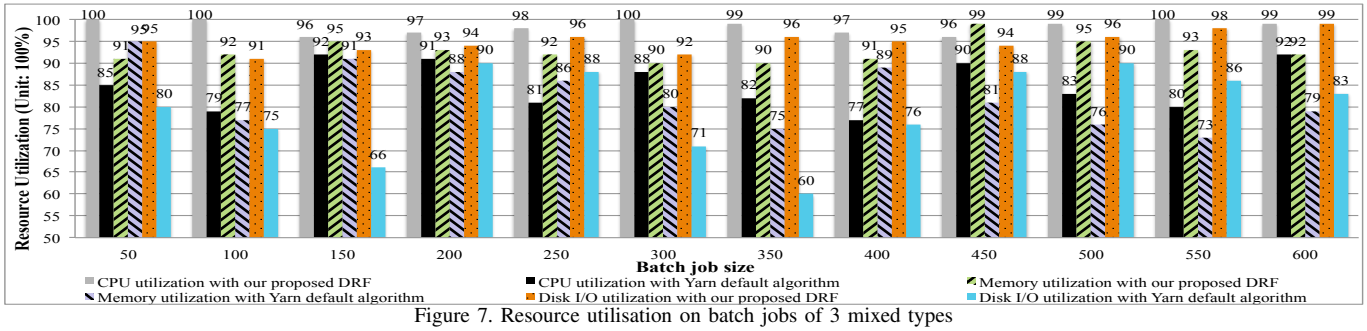

Figure 7. Resource utilisation on batch jobs of 3 mixed types

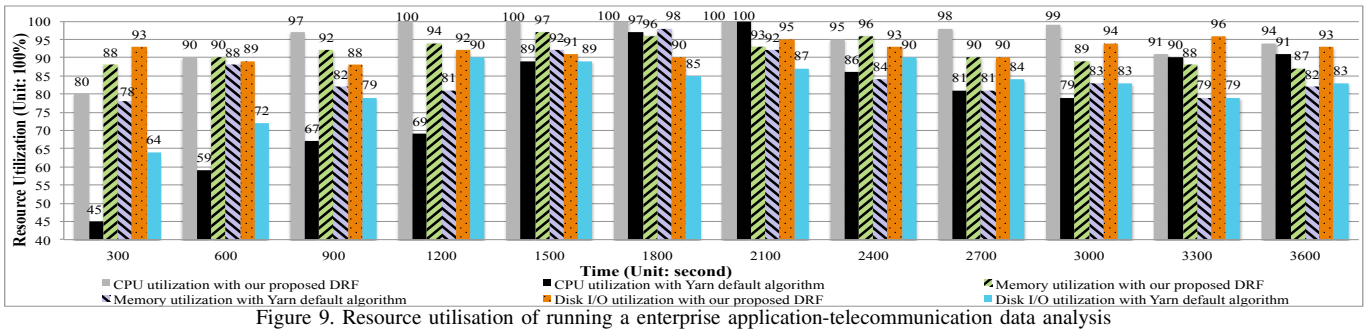

Figure 9. Resource utilisation of running a enterprise application-telecommunication data analysis

We can see our method's CPU utilisation is 17% higher and more steady except at the first testing point. In the beginning, image files are loaded into memory, so CPU and I/O usage are both not too high. Yet, the default algorithm's CPU usage is unstable, suddenly changing from 100% to 45%. Memory utilisation of our method is 82.75% higher than that of default algorithm (73.33%). Our DRF's I/O utilisation is 68.25% and not very steady but still better and flatter than default one (49.25%). This application prefers consuming CPU resource.

3) Use case 3: As this application is memory-intensive, the memory is the highest in all of these 3 resources in both our DRF and the default algorithm. Our algorithm's memory, CPU and I/O utilisations are 88.03%, 81.33% and 69.42% respectively on average. In comparison, the default algorithm's memory, CPU and I/O usages are 76.58%, 69.75% and 64.58% respectivel,y as shown in Figure 11. In the beginning, memory utilisation for both algorithms is slightly lower (only approximately 45%) than those on other detecting points. Since at first, data is read from disks and disk I/O utilisation of our DRF is very efficient (about 94.8%), which is 25% higher than the average value. In the default algorithm, we can see a similar phenomenon that I/O usage in the beginning is 16% higher than later. Our algorithm's memory utilisation is very high and steady (minimum is 88% and maximum is 100%) except 2 beginning points, but the default one is unsteady and unpredictable. For CPU utilisation, we see similar results. With time increasing, I/O resource utilisation reduces.

4) Use case 4: I/O utilisation is the largest for all resources in our algorithm. Our method's I/O usage is 86.75%, 11.25% higher than default one as depicted in Figure 12. In comparison, I/O performance and stability of our DRF is better. Our method's CPU and memory utilisations are 63.58% and 58.92%, however the default scheduler' CPU, memory, and I/O usage rates are only 49.50%, 52.42% and 75.50% respectively. The reason why these two algorithms' CPU and memory utilisations are not high and flat is that the application is I/O-intensive. This confirms that using '*Cgroup*' is an effective way to control CPU and I/O. The result is similar to that of the benchmark application results. Even in a large cloud application at the enterprise level, our DRF is able to maximise the resource utilisation and increase the throughput more desirably. Overall, our algorithm is generic and universal and was not developed for a particular domain.

VI. Summary

Our modified Dominant Resource Fairness (DRF) allocation algorithm with <CPU, memory, vdisk> enables all tenants to share the cloud system's resources including CPU, memory and disk I/O more fairly, reasonably and efficiently. We implemented our method for YARN and combined it with LINUX '*Cgroup*' controls to limit different resource usage of each tenant's jobs. It also considers data isolation on both logical and physical hardware layers to reduce interference with other tenants' workload. Our resource allocation mechanism provides different resource sharing degrees to satisfy different tenant share requirements. Our experimental results show that CPU utilisation can be increased at least 30%, and I/O utilisation up to 45%. It also addresses the problem when resource utilisation is becoming very unpredictable and erratic, as well as trying to avoid some tenants preempting too many resources. Our benchmark application experiments show that our method has significant utilisations improvements of 14% CPU, 8% memory, and 11% I/O. Even when running different kinds of applications, our scheduler delivers better performance, given that the algorithm has been designed for universal applications and focuses on the task level, rather than a specific application type. We will further work on scenarios where resource requirements are not given in advance and the requirements of tenants are varied. Combining various machine learning classifiers and adaptive algorithms together with our proposed DRF method could be a way to enhance its capabilities.

VII. Acknowledgement

This research is supported by a scholarship from Swinburne University of Technology, and in part by the City University of Hong Kong research funds (No. 7200354 and 7004222).

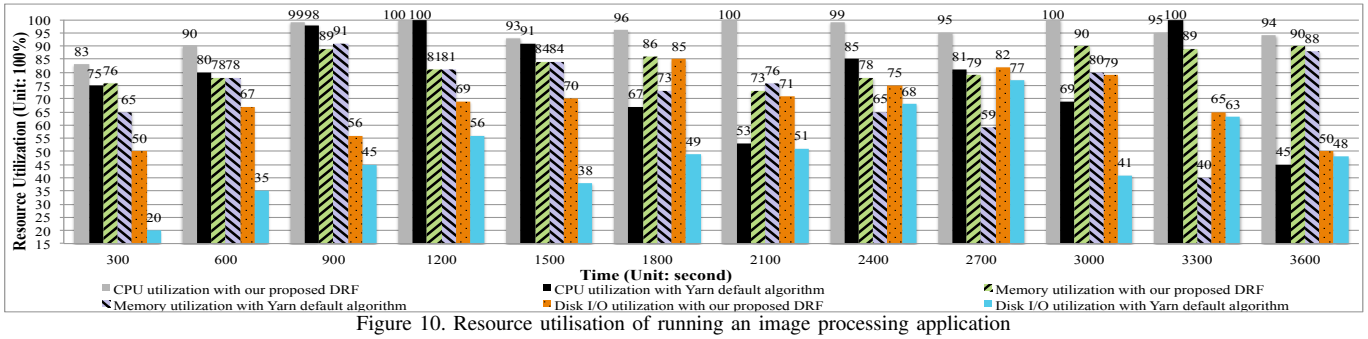

Figure 10. Resource utilisation of running an image processing application

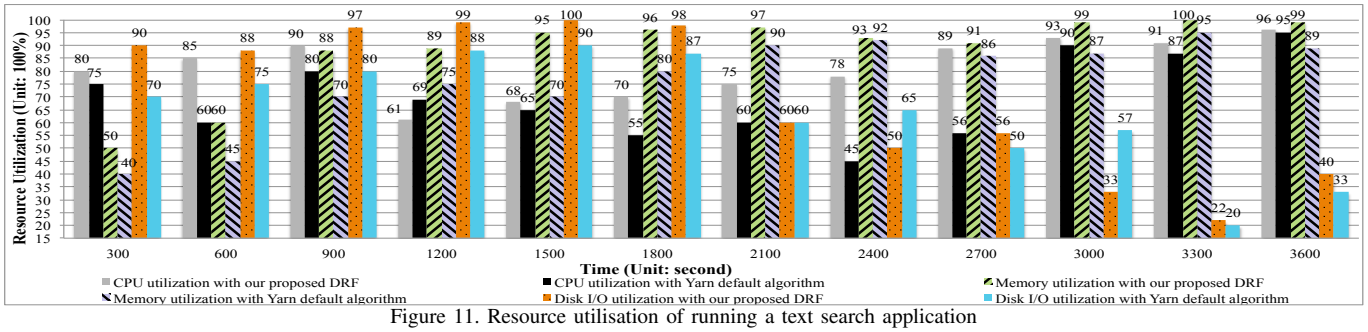

Figure 11. Resource utilisation of running a text search application

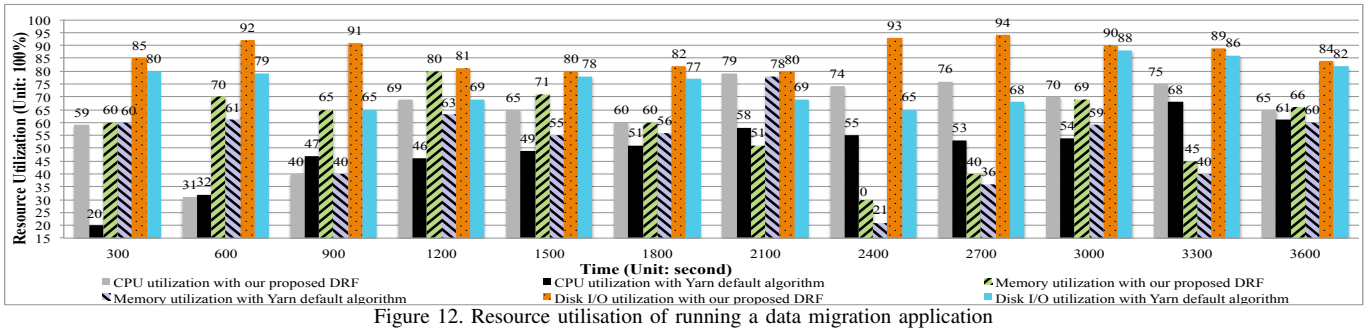

Figure 12. Resource utilisation of running a data migration application

REFERENCES

[1] G. E. Gonçalves, P. T. Endo, T. Cordeiro, Palhares et al., "Resource allocation in clouds: concepts, tools and research challenges," XXIX SBRC-Gramado-RS, 2011.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," Communications of the ACM, 2008, pp. 107–113.

[3] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in Proceedings of 8th ACM European Conference on Computer Systems, 2013, pp. 365–378.

[4] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza, "Dynamic resource allocation for database servers running on virtual storage." in FAST, vol. 9, 2009, pp. 71–84.

[5] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in NSDI, vol. 11, 2011, pp. 24–24.

[6] "Hortonworks data platform: system administration guides," in http://hortonworks.com, 2014.

[7] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks," in Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA, 2015, pp. 293–307.

[8] L. Tassiulas and S. Sarkar, "Maxmin fair scheduling in wireless networks," in INFOCOM. 21st Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE, 2002, pp. 763–772.

[9] "Max-min fairness," in http://en.wikipedia.org/wiki/Max-min_fairness.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, Konar et al., "Apache hadoop yarn: Yet another resource negotiator," in Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013, p. 5.

[11] T. White, Hadoop: The definitive guide. O'Reilly Media, 2012.

[12] "Dominant resource fairness on fair scheduler," in https://issues.apache.org/jira/secure/attachment/12581004/FairSchedulerDRFDesignDoc-1.pdf.

[13] "Cgroup," in http://en.wikipedia.org/wiki/Cgroups.

[14] "Kernel cgroup," in https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[15] "Blkio," in https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt.

[16] "Completely fair scheduler," in https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[17] M. T. Jones, "Inside the linux 2.6 completely fair scheduler," IBM Developer Works Technical Report, 2009.

[18] "Hadoop," in http://hadoop.apache.org.

[19] A. Murthy, V. K. Vavilapalli, D. Eadline, J. Markham, and J. Niemiec, Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2. Pearson Education, 2013.