Eunomia: A Performance-Variation-Aware Fair Job Scheduler With Placement Constraints For Heterogeneous Datacenters

Wei Zhou
University of Virginia
Charlottesville, USA
wz5ad@virginia.edu

K. Preston White
University of Virginia
Charlottesville, USA
kpwhite@virginia.edu

Hongfeng Yu
University of Nebraska-Lincoln
Lincoln, USA
hfyu@unl.edu

***Abstract*—Due to hardware upgrades and server consolidation, it is not uncommon to witness a few generations of servers deployed in the same datacenters. As a result, variants of fair job schedulers are proposed to enforce fairness for constrained jobs that have hardware or software constraints on task placement. However, the other important characteristics resulted from server heterogeneity, performance variation, is unfortunately overlooked by state-of-art fair job schedulers with placement constraints.**

In this paper, we propose *Eunomia*, a performance-variation-aware fair job scheduler, to address the unfairness issue due to performance variation in heterogenous clusters. Eunomia introduces a key metric, called *progress share*, which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Evaluation results show that Eunomia is able to deliver better share fairness compared with state-of-art schedulers without performance loss.

***Index Terms*—Big Data, Fair Scheduler, Placement Constraint**

I. INTRODUCTION

Recent workload analysis based on traces collected from Google compute clusters demonstrates the inherent nature of enormous heterogeneity of machine classes and variability of resource requirements from applications [1], [2]. It is common to see 3 to 5 generations of machines composed of up to 40 different configurations co-existing in an enterprise datacenter [1]–[4]. Server heterogeneity is an outcome of hardware upgrades and consolidation over time for achieving cost effectiveness. For cloud datacenters, cloud service providers also aim to provide differentiated services through variations of server configurations to attract users with various performance/price needs. For example, Amazon Web Services (AWS) offer a variety of instance types available to users, where the number of virtual CPUs (vCPUs) per instance ranges from 1 vCPU to 96 vCPUs and the clock speed of a vCPU ranges from 2.3GHz to 3.3GHz [5]. On the other hand, complexity and variability of application software impose big challenges to compute nodes in satisfying hardware and software constraints including, minimal number of cores, target micro-architecture, compatible OS kernel versions, specific libraries, etc. [1], [2].

This renders big data fair schedulers originally developed for homogeneous clusters in large-scale enterprise or cloud datacenters, difficult to enforce share fairness among different entities [6], [7] while satisfying hardware and software constraints. To this end, fair schedulers with placement constraints, that is, aiming at achieving fair shares in scheduling jobs on servers that meet the constraints specified by jobs, become a new hot research topic [4], [8]–[12]. Choosy [8] is the pioneer work of fair schedulers to support placement constraints, and its core idea has been extended to support server heterogeneity [9] and hierarchy of organizations [10]. However, none of state-of-art fair schedulers with placement constraints takes performance variation due to server heterogeneity into consideration, and is hard to achieve actual share fairness and quality of service.

In this paper, we propose Eunomia, a performance-variation-aware fair job scheduler with placement constraints for heterogeneous datacenters. Eunomia introduces *progress share fairness*, which is meant to equalize progress share of jobs as much as possible. Progress share of a job is defined as the ratio between the accumulated progress of scheduled tasks of a job, and the maximum accumulated progress of tasks that can run in the cluster if placement constraints are removed. In other words, the objective of Eunomia is to make execution of concurrently scheduled jobs with the same progress rate, regardless of performance variation due to hardware heterogeneity. As far as we know, Eunomia is the first job scheduler to reach share fairness for constrained jobs in the presence of performance variation among compute nodes.

We have implemented an Eunomia prototype and conducted quantitative evaluations via trace-driven simulations. Simulation results based on micro-benchmarks and Google traces shows that, Eunomia is able to deliver better share fairness compared with two state-of-art schedulers: Choosy [8] and TSF [4], without performance loss.

II. Related Work and Motivations

A. Fair Schedulers

Basically, a fair job scheduler aims to enforce fair sharing of computing resources in the cluster among the users. Guaranteeing scheduling fairness is important for job schedulers to support multiple tenancies in cloud datacenters. As an example, YARN has two built-in job schedulers: Capacity Scheduler and Fair Scheduler [13]. Both schedulers are used to share available resources in the cluster among multiple organizations, with capacity and fairness guarantees respectively. Capacity Scheduler partitions CPU and memory resources based on the capacity assigned to organizations, and maintains a job queue for each partition. Fair Scheduler is very similar to Capacity Scheduler, but it is meant to assign available resources to jobs fairly so that each job has an equal share of resources.

Generally, most of existing fair schedulers are based on the max-min fairness algorithm. Given each user has enough demand and equal share, it maximizes the lowest share first, then the second lowest, and then the third lowest, and so on. In such a policy, when max-min fairness is reached, increasing the share of a user will result in the decrease of the share of the others with equal or smaller allocations (Pareto efficiency) [7]. The attractive feature of the max-min fairness algorithm is to easily support weighted fairness in resource allocations. By assigning different weights to different users in max-min fairness, it is able to allocate resources to each user according to his/her share (equal share becomes one special case where every user has the same weight), and ensure a user's share regardless of the demand of other users. YARN's Fair Scheduler is based on the max-min fairness algorithm, and there has been a large body of literature on improving existing fair schedulers to enable them to adapt dynamic and various workloads and environments. Traditionally, fair schedulers take only one resource type, CPU, into considerations, and use the number of cores as the metric to determine the quantity of allocations for each user. DRF (Dominant Resource Fairness) is a generalization of the classical max-min fairness to multiple resource types [7], DRF defines "dominant share" as the maximum share of any resource type a user is allocated, and aims to maximize the minimum dominant share for all the users. DRFH is then proposed to extend the DRF idea to cloud environments with heterogeneous servers [9]. CMMF (Constrained Max-Min Fairness) and its online scheduler Choosy were proposed to extend the classic max-min fairness to support placement constraints [8]. TSF (Task-Share Fairness) was then proposed to extend the idea of Dominant Resource Fairness to support placement constraints in multiple-resource sharing environments [4]. The basic idea of TSF is to equalize the task share of each user and maximize the minimal task share first.

Although there are some fair schedulers of constrained jobs taking heterogeneity into considerations as aforementioned, they mainly focus on how to schedule jobs with demands of multiple types of resources and supplies of various hardware configurations. Eunomia is the first to take performance variation due to server heterogeneity into account and uses the key metric of *progress share* to reach better share fairness.

B. Performance Variability

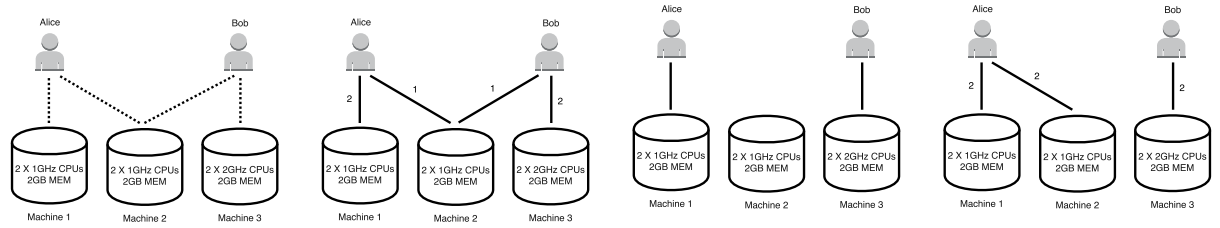
Server heterogeneity has been commonly observed and recognized in production datacenters. Usually, several generations of machines with different hardware configurations may coexist in the same cluster [1], [2]. A subset of machines may even be equipped with GPUs for special tasks like visualization, high-performance computing, or machine learning. For example, processors used in Amazon Web Services (AWS) are a variety of Intel Xeon CPUs including E5-2680v2, E5-2686v4, E5-2670, E7-8880v3, E5-2670v2, E5-2676v3 series, and processors' clock speeds range from 2.3GHz to up to 3.3GHz [14]. For another example, SSD storage provided in AWS ranges from 1 $\times$ 4GB SSD to 8 $\times$ 80GB SSDs while its HDD storage ranges from 3 $\times$ 2TB HDDs to 24 $\times$ 2TB HDDs. Various combinations of computation and I/O configurations lead to performance variability in heterogeneous clusters.

C. An Illustrative Example

Simply applying fair schedulers for constrained jobs to heterogeneous clusters cannot reach actual share fairness without taking performance variation into account. Moreover, it is not uncommon to witness the cases like the scheduling systems were manipulated to gain advantages by greedy users in large companies.

We give an example of naive fair schedulers for constrained jobs that do not consider performance variation in Figure 1a. As shown in this figure, we assume there are 3 machines. Machines 1 and 2 have the same hardware configuration: two 1GHz CPUs and 2GB MEM, denoted as $<2 \times$ 1GHz CPUs, 2GB MEM$>$. Machine 3's configuration is $<2 \times$ 2GHz CPUs, 2GB MEM$>$. Due to software constraints, Alice's job can only run on Machines 1 and 2, while Bob's job can only run on Machines 2 and 3, as shown in Figure 1a where dotted lines denote placement constraints. The tasks of the two jobs have the same demand: $<$1 CPU, 1GB MEM$>$.

As shown in Figure 1b, because Alice and Bob have the equal share of the cluster, Alice can run 3 tasks (2 on Machine 1 and 1 on Machine 2), while Bob also can run 3 tasks (2 on Machine 3 and 1 on Machine 2). This seems a fair resource allocation, since Alice and Bob run the same number of tasks on the cluster. However, it is clear that 2 of Bob's 3 tasks are running on the faster CPU, and none of Alice's tasks are running on the faster CPU. If both jobs have the same number of tasks and it is assumed that task execution time is inversely proportional to CPU clock speed (note it is not always a realistic assumption, we just want to simplify the assumption in this example), Bob benefits more from the allocation because his job can be completed much earlier than Alice.



(a) User/machine assumption (b) Naive fair allocation (c) After 1st round of allocations in Eunomia (d) Final allocations in Eunomia

Fig. 1: An illustrative example of resource allocations by naive fair scheduler and Eunomia

III. EUNOMIA

A. Basic Idea

To this end, we propose *Eunomia*, a performance-variation-aware fair scheduler, which takes performance variation due to server heterogeneity into considerations, and aim to equalize the progress share for each user. In Eunomia, progress share is computed as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Accumulated task progress is defined as the sum of the product of the task progress on a type of servers and the allocated number of the servers of the same type. Progress share can be treated as work slowdown of a job due to resource sharing and placement constraints. Assume task execution time on $<1 \times$ 1GHz CPU, 1GB MEM$>$ is p, and task execution time on $<1 \times$ 2GHz CPU, 2GB MEM$>$ is $p/2$, Table I gives the per-CPU task progress as a function of different types of nodes. If Alice is allocated with 2 $<1 \times$ 1GHz CPU, 1GB MEM$>$ and 1 $<1 \times$ 2GHz CPU, 2GB MEM$>$, then the accumulated task progress of Alice's job $= 2 \times 1 + 1 \times 2 = 4$.

TABLE I: CPU task progress matrix

Server Type	Alice's Task Progress	Bob's Task Progress
$<$1GHz CPU, 1GB MEM$>$	1	1
$<$2GHz CPU, 2GB MEM$>$	2	2

According to the progress share of a user, Eunomia applies the max-min fair allocation, that is, maximizes the lowest progress share first, then the second lowest, and then the third lowest, and so on. Let's continue to use the example in 1a to illustrate how the allocations are undertaken in Eunomia.

As shown in Figure 1c, assume Machine 1 is allocated to Alice and Machine 3 is allocated to Bob respectively. Then, Alice's progress share is $(2\times1)/(2\times1+2\times1+2\times2) = 2/8$, Bob's progress share is $(2 \times 2)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 4/8$. It is clear that Alice has the lowest progress share. Then 1 $<1 \times$ 1GHz CPU, 1GB MEM$>$ on Machine 2 is further allocated to Alice. Then Alice's progress share is $(3\times1)/(2\times 1 + 2 \times 1 + 2 \times 2) = 3/8$, which is still the lowest progress share. Then one more $<1 \times$ 1GHz CPU, 1GB MEM$>$ on Machine 2 is allocated to Alice. Then Alice's progress share is $(4 \times 1)/(2 \times 1 + 2 \times 1 + 2 \times 2) = 4/8$, and both Alice and Bob have the same progress share now, as shown in Figure 1d.

B. Offline and Online Eunomia Algorithm

Figures 1c and 1d give an intuitive example to demonstrate how resources are allocated by Eunomia with progressive filling. The basic idea of *progressive filling* is to incrementally reach target fair sharing through multiple rounds, and it is widely used in various max-min fair job schedulers including Choosy [8] and TSF (Task Share Fairness) [4]. In the first round, Eunomia computes and equally raises progress shares for all the users based on their resource allocations until the maximum progress share is achieved. Then the users whose progress shares cannot be further raised are treated "inactive" users, and their resource allocations are frozen. In the second round, Eunomia continues to further recompute and equally raise progress shares of the remaining active users while keeping those of the inactive user(s) unchanged. Eunomia repeats the process round by round until all the users become inactive. The progressive filling method is considered an offline algorithm since it is impractical to implement due to its prohibitively high computation overheads. Table II and Algorithm 1 give terminology and a formalized algorithmic description of the offline Eunomia algorithm with progressive filling. Note that p_i is the normalized performance vector which are determined by the task execution time of user i on different machines.

We now illustrate our Eunoima algorithm as shown in Algorithm 1. In each round, Eunoima first finds the maximum progress share s^t that is achieved for all active users by solving a linear programming problem. In particular, the linear programming problem has a non-negative variables x_{im}, which denotes the number of tasks of user i scheduled on machine m, and three types of constraints: machine resource capacity constraints, user constraints for active users and inactive users, respectively. The active users' constraints (1) ensure that in each round, each active user's progress shares is raised equally (See Line 37). The inactive users' constraints (2) ensure that a user will not decrease his/her allocated number of tasks when he/she becomes inactive (See Line 38). And the resource capacity constraints (3) ensure that all users' allocated number of tasks on each machine does not exceed the machine's capacity (See Line 39).

Algorithm 1 Offline Eunomia Scheduler Using Progressive Filling

procedure EUNOMIA(L_m, p_i, d_i, c_i, w_i)
  $t \leftarrow 1$ ▷ Current round
  $\mathcal{U}^t \leftarrow \{1, 2, ..., N\}$ ▷ Initialize active user set
  for $i \in \mathcal{U}^t$ **do**
    $x_i \leftarrow 0$ ▷ Initialize tasks number scheduled for each user
  end for
  while true do
    $(\{x_{im}\}, s^t) \leftarrow$ LP($t, \mathcal{U}^t, L_m, p_i, d_i, c_i, w_i$)
    $(\{x_i\}, \mathcal{U}^{t+1}) \leftarrow$ SATURATED($t, s^t, \mathcal{U}^t, L_m, p_i, d_i, c_i, w_i$)
    if $\mathcal{U}^t = \emptyset$ **then** ▷ All users saturated?
      return $\{x_{im}\}$ ▷ Return number of tasks scheduled on each node for each user
    end if
  end while
  $t \leftarrow t$+1
end procedure
procedure SATURATED($t, s^t, \mathcal{U}^t, L_m, p_i, d_i, c_i, w_i$)
  $\mathcal{V}^t \leftarrow \emptyset$ ▷ Inactive user set after round t
  for $i \in \mathcal{U}^t$ **do**
    for $j \in \mathcal{U}^t \backslash \{i\}$ **do**
      $x_j \leftarrow \sum_{m=1}^{M} x_{jm}$ ▷ Freeze all but i
    end for
    $(\{x_{im}^z\}, s^z) \leftarrow$ LP($t, \{j\}, L_m, p_i, d_i, c_i, w_i$)
    if $s^z == s^t$ **then** ▷ If progress share cannot be increased
      $\mathcal{V}^t \leftarrow \mathcal{V}^t \cup \{i\}$ ▷ User i becomes inactive
    end if
    for $j \in \mathcal{U}^t \backslash \{i\}$ **do**
      $x_j \leftarrow 0$ ▷ Unfreeze number of tasks of active users
    end for
  end for
  for $i \in \mathcal{U}^t$ **do**
    $x_i \leftarrow \sum_{m=1}^{M} x_{im}$ ▷ Freeze number of tasks of inactive users
  end for
  return $(\{x_i\}, \mathcal{U}^t \backslash \mathcal{V}^t)$
end procedure
procedure LP($t, \mathcal{U}^t, L_m, p_i, d_i, c_i, w_i$)
  maximize s^t subject to:

$$\frac{1}{w_i \sum_{m=1}^{M} f_{im} p_{im}} \sum_{m=1}^{M} x_{im} p_{im} c_{im} = s^t, i \in \mathcal{U}^t \quad (1)$$

$$\sum_{m=1}^{M} x_{im} c_{im} \geq x_i, i \notin \mathcal{U}^t \quad (2)$$

$$\sum_{i=1}^{N} x_{im} d_{ir} \leq l_{mr}, m \in [1, M], r \in [1, R] \quad (3)$$

  return $(\{x_{im}\}, s^t)$
end procedure

After all active users obtain the maximum progress share s^t in round t, Eunomia does a test to determine which users remain active. It solves the same linear programming problem mentioned above for each active user i by keeping all users except i at their previous round's allocated number of tasks, while maximizing the progress share of user i. If it is impossible to increase user i's progress share, then user i is saturated and becomes inactive, and his/her allocated number of tasks is frozen.

The offline Eunomia algorithm with progressive filling needs to recompute and raise progress shares for the users remaining active in each round. However, it is impractical to implement due to its prohibitively high computation overheads. In practice, resource allocations and job scheduling only come into play when (1) a new job arrives and at least one server meets task resource demands of the job; and (2) a server completes one task and the resource is freed for re-allocation. Similar to other fair schedulers, we develop a simple online Eunomia algorithm, that is, whenever resources on a server become available, Eunomia allocates the resources to the user with the current lowest progress share, whose constraints can be met by the server. We measure the fairness and performance of the online Eunomia algorithm in the following.

TABLE II: Terminology in the offline Eunomia algorithm with progressive filling

Notation	Description
N	The total number of users
M	The total number of machines in the cluster
R	The total types of resources
L_m	Resource capacity vector, where l_{mr} is the capacity of resource r on machine m
p_i	Normalized performance vector of user i, where p_{im} is the normalized performance of machine m for user i
d_i	Normalized user demand vector of user i, where d_{ir} is the demand of resource r for a task of user i
c_i	User constraint vector, where $c_{im} = 1$ if machine m can run tasks of user i. Otherwise, $c_{im} = 0$
w_i	User weight where w_i is the weight of user i
x_{im}	x_{im} is the number of tasks of user i scheduled on machine m
f_{im}	f_{im} is the number of tasks of user i scheduled on machine m in the hypothetical case where the cluster is monopolized by user i without any constraints
s_i	s_i is the progress share for user i where $s_i = \frac{x_i}{f_i \times w_i}$
U	Active user set in round t
V	Inactive user set after round t

TABLE III: Node configuration in the micro-benchmark

Node Configuration	Type-1 nodes	Type-2 nodes	Type-3 nodes	Type-4 nodes
Number of Nodes	5	5	5	5
Normalized Performance	1.0	1.5	2.0	3.0
Number of Cores	4	4	4	4
Memory (GB)	4	4	4	4

IV. Experimental Results

A. Experimental setup

We develop an event-driven job scheduler simulator to conduct fairness and performance evaluations of Eunomia. This simulator takes job traces as input and is able to simulate the entire process and resulting events of job schedulers, from job arrival, job queueing, dispatching tasks to nodes, receiving completion from nodes, and job departure. Currently this simulator is able to simulate two state-of-art fair schedulers of constrained jobs: Choosy and TSF, as well as our proposed Eunomia. It is also highly configurable, can be easily used to simulate a cluster consisting of hundreds or thousands of nodes.

B. Micro-benchmark experiment results

Micro-benchmark experiments are to demonstrate how Eunomia schedules constrained jobs on heterogeneous nodes with performance variation, and compare the fairness behaviors of state-of-art fair schedulers like Choosy and TSF with the proposed Eunomia. In micro-benchmark experiments, we simulate a cluster consisting of 4-types of nodes with different normalized performance. There are 20 nodes in total, that is, 5 nodes per type. The detailed node configuration information is depicted in Table III.

In this micro-benchmark evaluation, we simulate 4 jobs with nearly the same configuration arriving at the same time, except the last job explicitly places constraints on type-4 nodes. This is to compare how the fair job schedulers allocate resources if a user intentionally places job constraints on high-performing nodes and wants to take advantage of this false job constraint. The detailed job configuration information is depicted in Table IV.

TABLE IV: Job configuration in micro-benchmark experiment

Job configuration	Job 1	Job 2	Job 3	Job 4
Start time(s)	0	0	0	0
Number of tasks	1000	1000	1000	1000
Demand of cores per task	1	1	1	1
Demand of memory per task	1GB	1GB	1GB	1GB
Mean task execution time on Type-1 nodes(s)	2	2	2	2
Nodes meeting constraints	All types of nodes	All types of nodes	All types of nodes	Type-4 nodes

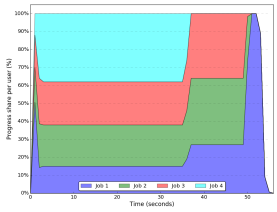


Fig. 3: Progress shares for 4 jobs over time under the TSF scheduler

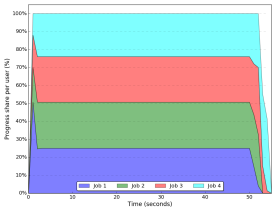
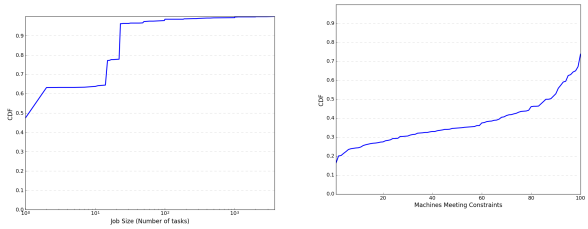


Fig. 4: Progress shares for 4 jobs over time under the Eunomia scheduler

Figures 3 and 4 shows that the arrival and completion of 4 jobs under TSF and Eunomia schedulers respectively (We omit the result of Choosy because it is very similar to TSF). Figure 3 clearly demonstrates for TSF Job 4 is completed much earlier than other jobs. This is because Job 4 obtains the most progress share (40%) until it is completed. It indicates that TSF fails to achieve the fairness without considering performance variation due to server heterogeneity. Figure 4 depicts how Eunomia schedules the same 4 jobs. One can see that each job obtains equal progress share during their execution time, and all 4 jobs are completed nearly at the same time. It is evident that Eunomia is able to deliver better fairness than TSF under server-heterogeneous environments and resistant to false job placement constraint requirements from greedy users.

C. Macro-benchmark experiment results

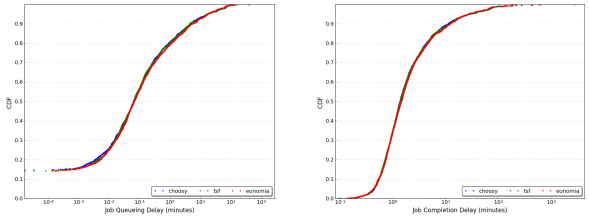
Macro-benchmark experiments are meant to validate the performance of the proposed Eunomia job schedulers. Note that the main goal of a fair job scheduler is to deliver the guaranteed fairness instead of performance improvement, so the purpose of macro-benchmark experiments is to show the performance impact of Eunomia compared with other state-of-art fair job schedulers. To this end, we take publicly available Google cluster traces as input, synthesize and feed the workload to a simulated cluster consisting of 100 nodes. The original Google cluster traces cannot be directly used in the



(a) Distribution of job size

(b) Distribution of machines meeting job constraints

Fig. 5: Distribution of job size and machines meeting job constraints in the synthesized trace



(a) Distribution of job queueing delay

(b) Distribution of job completion delay

Fig. 6: Distribution of job queueing and completion delay

simulation because they are a set of sampled job events, task events, machine events, machine attributes, task constraints, etc., and synthesization work needs to be done to sample and extract the job information and compose the needed job traces including job arrival time, number of tasks, number of CPU cores required, and amount of memory required. The synthesized workload consists of 63,976 tasks across 2,888 jobs in 5,000 seconds. Figure 5a shows the distribution of job sizes. We also sample and synthesize the needed node information from Google cluster traces, for example, number of CPU cores and amount of memory for each node. We follow the latest way proposed by Sharma *et al.* [2] to synthesize job and node constraints. Figure 5b shows the distribution of nodes meeting job constraints. One can see about 18% of jobs can be run at any nodes while 50% of nodes can run 35% of jobs. In addition, we categorize 100 nodes into 10 types, and each type has different normalized performance ranging from 1.0 to 3.25, with a step of 0.25.

Figures 6a and 6b show the CDF distribution of job queueing delay and job completion delay of Choosy, TSF, and Eunomia respectively. Job queueing delay is defined as the duration between the arrival time of a job and the time when its first task of the job is scheduled. Job completion delay is defined as the duration between the arrival time of a job and the time when its last task of the job is completed. Since the cluster is idle when the simulation starts, we omit the performance results of the first 288 jobs (10% of total jobs) and consider it as a "warm-up" period of the cluster. One can see that Eunomia achieves nearly the same job queueing delay and job completion delay as Choosy and TSF do. It implies that Eunomia does not cause any performance loss compared with state-of-art fair schedulers of constrained jobs.

V. Conclusion

In this paper, we propose Eunomia, a performance-variation-aware fair job scheduler, to address the unfairness issue due to performance variation in heterogenous clusters. Eunomia introduces a key metric, called *progress share*, which is defined as the ratio between the accumulated task progress given the current allocation and the accumulated task progress if the user can monopolize the cluster. Eunomia aims to equalize progress share of jobs as much as possible, so as to achieve the same slowdown of jobs from different users due to resource sharing and placement constraints, regardless of performance variation. Simulation results show that Eunomia is able to deliver better share fairness compared with state-of-art schedulers without performance loss.

References

[1] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *3rd ACM Symposium on Cloud Computing (SoCC '12)*, 2012.

[2] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *2nd ACM Symposium on Cloud Computing (SoCC '11)*, 2011.

[3] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2002.

[4] Wei Wang, Baochun Li, Ben Liang, and Jun Li. Multi-resource fair sharing for datacenter jobs with placement constraints. In *2016 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '06)*, 2016.

[5] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/.

[6] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Gravesy, Jason Lowey, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen OMalley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *4th Annual Symposium on Cloud Computing (SoCC '13)*, 2013.

[7] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX conference on Networked systems design and implementation (NSDI '11)*, 2011.

[8] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *8th ACM European Conference on Computer Systems (EuroSys '13)*, 2013.

[9] Wei Wang, Baochun Li, and Ben Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *33rd Annual IEEE International Conference on Computer Communications (INFOCOM '14)*, 2014.

[10] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *2013 ACM Symposium on Cloud Computing (SoCC '13)*, 2013.

[11] Shanjiang Tang, Bu-Sung Lee, Bingsheng He, and Haikun Liu. Long-term resource fairness: Towards economic fairness on pay-as-you-use computing systems. In *28th ACM International Conference on Supercomputing (ICS '14)*, 2014.

[12] Haikun Liu and Bingsheng He. Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '14)*, 2014.

[13] Tom White. *Hadoop: The Definitive Guide, 4th Edition*. OReilly Media Inc., 2015.

[14] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/.