MISTA 2015

Resource-Aware Scheduling for Data Centers with Heterogenous Servers

Tony T. Tran[+] · Peter Yun Zhang[°] · Heyse Li[+] · Douglas G. Down[*] · J. Christopher Beck[+]

Abstract This paper presents an algorithm for resource-aware scheduling of computational jobs in a large-scale heterogeneous data center. The algorithm aims to allocate different machine configurations to job classes to attain an efficient mapping between job resource request profiles and machine resource capacity profiles. We propose a three-stage algorithm. The first stage uses a queueing model that treats the system in an aggregated manner with pooled machines and jobs represented as a fluid flow. The latter two stages use combinatorial optimization techniques to take the solution from the first stage and apply it to a more accurate representation of the data center. In the second stage, jobs and machines are discretized. A linear programming model is created to obtain a solution to the discrete problem that maximizes the system capacity. The third and final stage is a scheduling policy that uses the solution from the second stage to guide the dispatching of arriving jobs to machines. Using Google workload trace data, we show that our algorithm outperforms a benchmark greedy dispatch policy. We find that our algorithm is able to provide mean response times up to an order of magnitude smaller than the benchmark dispatch policy. These results show that it is important to consider the heterogeneity of machine configuration profiles in making effective scheduling decisions.

1 Introduction

The cloud computing paradigm of providing hardware and software remotely to end users has become very popular with applications such as e-mail, Google documents, iCloud, and dropbox. Service providers employ large data centers to provide these

[+] Department of Mechanical and Industrial Engineering,
University of Toronto
E-mail: {tran, hli, jcb}@mie.utoronto.ca

[°] Engineering Systems Division
Massachusetts Institute of Technology
E-mail: pyzhang@mit.edu

[*] Department of Computing and Software
McMaster University
E-mail: downd@mcmaster.ca

applications. As the demand for computational resources increases, the supply of services must efficiently scale. Yet, data centers represent a significant capital investment. Not only are servers for a data center expensive, maintaining and running a data center is a substantial investment. Due to the significant cost of these machines, many data centers are not purchased as a whole at one time, but rather built incrementally, adding machines in batches. Data center managers may choose machines based on the price-performance trade-off that is economically viable and favorable at the time [21]. Therefore, it is not uncommon to see data centers comprised of tens of thousands of machines, which are divided into ten or so different machine configurations, each with a large number of identical machines.

Under heavy loads, submitted jobs may have to wait for machines to become available before starting processing. These delays can be significant and can become problematic. Therefore, it is important to provide scheduling support that can directly handle the varying workloads and differing machines so that efficient routing of jobs to machines can be made. We study the problem of scheduling jobs onto machines such that the multiple resources available on a machine (e.g., processing cores and memory) can handle the assigned workload in a timely manner.

We develop an algorithm to schedule jobs on a set of heterogeneous machines to minimize mean job response time, the time from when a job enters the system until it starts processing on a machine. The algorithm consists of three stages. In the first stage a queueing model is used. Here, the system is represented at a very high level with resources and jobs both pooled. In each successive stage, a finer system model is used, such that in the third stage we generate explicit schedules for the actual system. Our experiments are based on job traces from one of Google's compute clusters [18] and show that our algorithm significantly outperforms a natural greedy policy that attempts to minimize the response time of each arrival.

The contributions of this paper are:

– The introduction of a hybrid queueing theoretic and combinatorial optimization scheduling algorithm for a data center, which efficiently maps job resource request profiles to different machine resource capacities.
– An extension to the allocation linear programming (LP) model presented in [3] and used for distributed computing in [2] to a data center that has multiple machines with multi-capacity resources.
– An empirical study of our scheduling algorithm on real workload trace data, which serves as a proof-of-concept of our proposed algorithm.

The rest of the paper is organized into a definition of the data center scheduling problem in Section 2, related work on data center scheduling in Section 3, a presentation of our proposed algorithm in Section 4, and experimental results in Section 5. Section 6 concludes our paper along with some plans for future work.

2 Problem Definition

The data center of interest is one that is comprised of many independent servers (also referred to as machines). We are interested in dealing with a server cluster that has on the order of tens of thousands of machines. These machines are not all identical; the entire machine population is divided into different configurations denoted by the set M. Machines belonging to the same configuration are identical in all aspects.

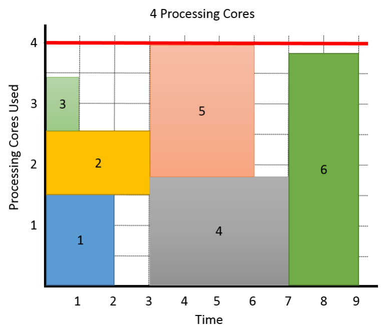


Fig. 1: Processing cores resource consumption profiles

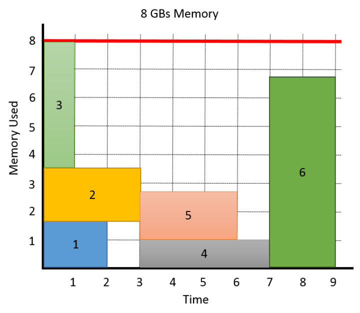


Fig. 2: Memory resource consumption profiles

We classify a machine configuration based on its resources. For example, machine resources may include the number of processing cores and the amount of memory, disk-space, and bandwidth. For our study, we generalize the system to have a set of resources, R, which are limiting resources of the data center. Each machine configuration is defined by the capacity of each resource available in the machines belonging to that configuration. A machine of configuration $j \in M$ has c_{jl} amount of resource $l \in R$ and within a configuration j there are n_j identical machines.

In our data center scheduling problem, jobs must be assigned to the machines with the goal of minimizing the mean response time of the system. We assume that jobs are assigned immediately as they arrive and the assignment cannot be changed. Jobs arrive to the data center dynamically over time. Times between arrivals are independent and identically distributed (i.i.d.). Each job belongs to one of a set of K classes where the probability of an arrival being of class k is α_k. A distribution of resource requirements for a job is defined by the class of the job. We denote the expected amount of resource of type l required by a job of class k as r_{kl}. The processing times for jobs in class k on a machine of configuration j are assumed to be i.i.d. with mean $\frac{1}{\mu_{jk}}$. The associated processing rate is thus μ_{jk}.

Each job is processed on a single machine. However, a machine can process many jobs at once, as long as the total resource usage of all concurrent jobs does not exceed the capacity of the machine. Figures 1 and 2 depict an example schedule of six jobs on a machine with two limiting resources: processing cores and memory. Here, the x-axis represents time and the y-axis is the amount of resource used. The machine has 4 processing cores and 8 GBs of memory. Note that the start and end times of each job are the same in both figures. This represents the job concurrently consuming resources from both cores and memory during its processing time.

Any jobs that do not fit within the resource capacity of a machine must wait until sufficient resources become available. We assume there is a buffer of infinite capacity for each machine where jobs can queue until they begin processing. Figure 3 illustrates the different states a job can go through in its lifetime. Each job begins outside the system and joins the data center once submitted. At this point, the job must be dispatched to one of the machines. This machine may or may not be immediately available for the job. The job must wait in the queue if there are insufficient resources, but can immediately start running if the required resources are available. If the job must join

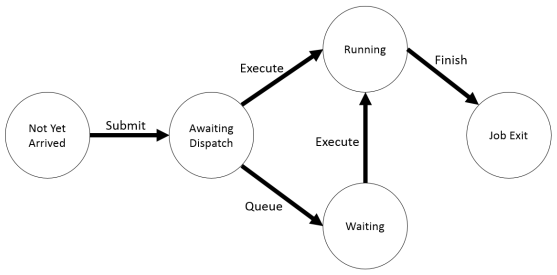


Fig. 3: Stages of job lifetime.

the queue, then it will start running on the machine when resources free up and the job has priority to start processing. Finally, after being processed, the job will exit the data center.

3 Related Work

Scheduling in data centers has received significant attention in the past decade. Many works consider cost saving through decreased energy consumption from lowering thermal levels [22,23], powering down machines [4,7], or geographical load balancing [13, 14]. These works often attempt to minimize costs or energy consumption while maintaining some guarantees on response time and throughput.

The literature on schedulers for distributed computing clusters has focused heavily on *fairness* and *locality* [11,19,24]. Optimizing these performance metrics leads to equal access of resources for different users and the improvement of performance by assigning tasks close to the location of stored data in order to reduce data transfer traffic. Locality of data has been found to be crucial for performance in systems such as MapReduce, Hadoop, and Dryad. Our work does not consider data transfer or equal access for different users. The works looking at *fairness* and *locality* also differ from our work in that our model focuses on the heterogeneity of machines with regard to resource capacity and how the mix of jobs that may be concurrently processed on a machine is a non-trivial decision.

Ghodsi et al. [8] and Grandl et al. [9] look at scheduling a system with multiple multi-capacity resources (e.g., CPU, memory, disk storage, and bandwidth). Ghodsi et al. [8] propose a scheduling policy, Dominant Resource Fairness, that aims to fairly share resources to each user based on their dominant resource. A dominant resource for each user is found by first normalizing resource requirements using the maximum capacity of the resource over all machines and then taking the resource that has the largest normalized requirement. For example, if a user requests two cores and two GB of memory and the maximum number of cores and memory on any system is four cores and eight GB, the normalized values would be 0.5 cores and 0.25 memory. The dominant resource for the user would thus be cores. Each user is then given a share of the resources such that the proportion of dominant resources for each user is equal to the dominant resource share of others. Note that this may compare resources of different types as the consideration is based on a user's dominant resource. Grandl et al. [9] study a similar problem, but emphasize the efficient packing of jobs onto machines. They propose the Tetris scheduler, which considers a linear combination of two scoring functions: packing jobs onto machines to best fit the remaining resources, and *least remaining work first* that looks at the remaining work (duration times resource requirements) of a job. The first score favours large jobs, while the second favours small jobs. Tetris chooses the next job to process based on the job with the maximum score. They compare Tetris against Dominant Resource Fairness and show that focusing on fairness alone can lead to poor performance. Their work shows the importance of considering efficient resource allocation, an issue that has had more attention recently. However, to effectively use Tetris, a system manager must tune several parameters to customize the job score for their application. Based on the job score employed, Tetris may over-prioritize large or small jobs and thus starve jobs that do not have high scores by constantly introducing new jobs with higher priority. A comparison of our proposed algorithm and the Tetris scheduler is a key area for future work.

Kim et al. [12] study dynamic mapping of jobs to machines in a heterogeneous environment. Jobs have varying priorities and soft deadlines. They find that two scheduling heuristics stand out as the best performers: *Max-Max* and *Slack Sufferage*. In *Max-Max*, a job assignment is made by greedily choosing the mapping that has the best fitness value based on the priority level of a job, its deadline, and the job execution time. *Slack Sufferage* chooses job mappings based on which jobs suffer most if not scheduled onto their "best" machines. Al-Azzoni and Down [2] schedule jobs to machines using an allocation LP to efficiently pair job classes to machines. The solution of the LP problem maximizes the system capacity and guides the scheduling rules to reduce the long-run average number of jobs in the system. Further, they show that their heuristic policy is guaranteed to be stable if the system can be stabilized. Rasooli and Down [20] extend the allocation LP model to address a Hadoop framework. They compare their work against the default scheduler used in Hadoop and the *Fair-Sharing* algorithm and show that their algorithm greatly reduces the response time, while maintaining competitive levels of fairness with Fair-Sharing. These papers focus on job execution time as the key defining characteristic in machine heterogeneity and do not consider multi-capacity resources of machines.

Chang et al. [5] consider a grid computing system where clusters of resources have varying computing speeds and the bandwidth capacities between clusters are different. The authors develop a scoring algorithm that maps jobs to resources based on the bandwidth availability and cluster load. Maguluri et al. [17] examine a cloud computing cluster where virtual machines are to be scheduled onto servers. Virtual machines require some amount of CPU, memory, and storage space that must fit onto the servers they have been assigned to. Their work assumes that there are different types of virtual machines: *Standard*, *High-Memory*, and *High-CPU*. Each virtual machine type has specified resource requirements and different instances of virtual machines within a single type do not differ. Based on these requirements and the capacities of the servers, the authors determine all possible combinations of virtual machines that can concurrently be placed onto each server. A preemptive algorithm is presented that uses the defined virtual machine combinations. They show that their algorithm is throughput-optimal. An alternative, non-preemptive algorithm is proposed that is close to throughput optimal. The algorithm works by choosing at the beginning of a time slot the mix of virtual machine types on each server to maximize the amount of work that can be done for that time slot. An extension to their work was later done to prove a queue-length optimal algorithm for the same problem in the heavy traffic regime [16]. They propose a routing algorithm that assigns jobs to servers with the shortest queue (similar to our greedy algorithm presented in Section 5.2) and a mix of virtual machines to assign to a server based on the same reasoning proposed for their throughput optimal algorithm. These works differ from our work since virtual machine types have predetermined resource requirements. Therefore, it is known exactly how virtual machine types will fit on a server without having to reason online about each assignment individually based on their specific requirements. Because the virtual machine sizes are set, inefficiencies due to fragmentation are not a concern as they are in our system. However, resource wastage due to fragmentation still exists from virtual machines not completely filling server capacities. Furthermore, fragmentation occurs *inside* the virtual machine as well since jobs may not use the full resources of a virtual machine type and will then occupy more resources (the size of a virtual machine) than required.

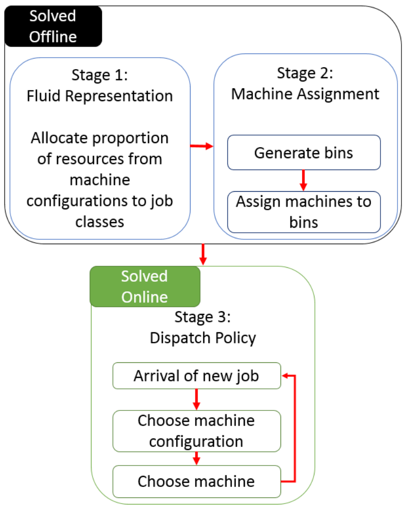
4 Data Center Scheduling

The proposed algorithm, Long Term Evaluation Scheduling (LoTES), is a three-stage queueing-theoretic and optimization hybrid approach. Figure 4 illustrates the overall scheduling algorithm. The first two stages are performed offline and are used to guide the dispatching algorithm of the third stage. The dispatching algorithm is responsible for assigning jobs to machines and is performed online. In the first stage, we use techniques from the queueing theory literature, which represent the data center as a fluid model where incoming jobs can be considered in the aggregate as a continuous flow. We extend the allocation LP model presented by Andradóttir et al. [3] to account for multiple resources. The allocation LP is used to find an efficient allocation of machine resources to job classes. In the second stage, a machine assignment LP model is used to assign specific machines to serve job classes using the results of the allocation LP. In the final stage, jobs are dispatched to machines dynamically as they arrive to the system.

4.1 Allocation LP

Andradóttir et al.'s [3] allocation LP was created for a similar problem but with a single unary resource per machine. The allocation LP finds the maximum arrival rate for a given queueing network such that stability is maintained. Stability is a formal property of queueing systems [6] that can informally be understood as the queue lengths in the system remaining bounded over time.

In our problem, there are $|R|$ resources which must be accounted for. We modify the allocation LP to accommodate these multiple resources. The model combines each machine's resources to create a single super-machine for each configuration. Thus, there will be exactly $|M|$ pooled machines (one for each configuration) with capacity $c_{jl} \times n_j$ for resource l. The allocation LP ignores resource fragmentation within the pooled machines. Fragmentation occurs when a machine's resource capacity cannot be fully utilized as a result of the currently available resources of a machine not being sufficient to admit a job, leaving resources unused with jobs waiting in queue. For example, if a configuration has 30 machines with 8 cores available on each machine and a set of jobs assigned to the configuration requires exactly 3 cores each, the pooled machine would have 240 processors that can process 80 jobs in parallel. However,



Solved Offline
Stage 1: Fluid Representation
Allocate proportion of resources from machine configurations to job classes
Stage 2: Machine Assignment
Generate bins
Assign machines to bins
Solved Online
Stage 3: Dispatch Policy
Arrival of new job
Choose machine configuration
Choose machine

Fig. 4: LoTES Algorithm.

only 2 jobs could be placed on each individual machine. Therefore, only 60 jobs can be processed in parallel. The effect may be further amplified when multiple resources exist as fragmentation could occur for each resource. The subsequent stages of the LoTES algorithm deal with the issue of fragmentation by treating each machine individually (see Section 4.2).

The extended allocation LP is given by (1)-(5) below.

$$\max \quad \lambda \tag{1}$$

$$\text{s.t.} \sum_{j \in M} (\delta_{jkl} c_{jl} n_j) \mu_{jk} \geq \lambda \alpha_k r_{kl} \qquad k \in K, l \in R \tag{2}$$

$$\frac{\delta_{jkl} c_{jl}}{r_{kl}} = \frac{\delta_{jk1} c_{j1}}{r_{k1}} \qquad j \in M, k \in K, l \in R \tag{3}$$

$$\sum_{k \in K} \delta_{jkl} \leq 1 \qquad j \in M, l \in R \tag{4}$$

$$\delta_{jkl} \geq 0 \qquad j \in M, k \in K, l \in R \tag{5}$$

Decision Variables

λ:	Arrival rate of jobs
δ_{jkl}:	Fractional amount of resource l that machine j devotes to job class k

The LP assigns the fractional amount of each resource that each machine pool should allot to each job class in order to maximize the arrival rate of the system, while maintaining stability. Constraint (2) guarantees that sufficient resources are allocated for the expected requirements of each class. Constraint (3) ensures that the resource profiles of jobs (i.e., the amount of each resource a job class is expected to request) are properly enforced. For example, if the amount of memory required is twice the number of cores required, the amount of memory assigned to the job class from a single machine configuration must also be twice that of the core assignment. The allocation LP does not assign more resources than available due to constraint (4). Finally, constraint (5) ensures the non-negativity of assignments.

Solving the allocation LP will provide δ^*_{jkl} values which tell us how we can efficiently allocate jobs to machine configurations. However, due to fragmentation, the allocation LP solution is only an upper bound on the achievable arrival rate of a system. The bound for the single unary resource problem is tight: Andradóttir et al. [3] show that utilizations arbitrarily close to one are possible. This is not possible when fragmentation occurs.

4.2 Machine Assignment

In the second stage, we use the job-class-to-machine-configuration results from the allocation LP to guide the choice of a set of job classes that each machine will serve. We are concerned with fragmentation and so treat each job class and each machine discretely, building specific sets of jobs (which we call "bins") that result in tightly packed machines and then deciding which bin each machine will emulate. This stage is still done offline and so rather than using the observed resource requirements of jobs, we use the expected values.

In more detail, recall that the δ^*_{jkl} values from the allocation LP provide a fractional mapping of the resource capacity of each machine configuration to each job class. Based on the δ^*_{jkl} values that are non-zero and the particular resource requests of jobs and the capacities of the machines, the machine assignment algorithm will first create job bins. A bin is any set of jobs that together do not exceed the capacity of the machine. A non-dominated bin is a bin which is not a subset of any other bin: if any additional job is added to it, one of the machine resource constraints will be violated. Figure 5 presents the feasible region for an example machine. Assume that the machine has one resource (cores) with capacity 7. There are two job classes, job class 1 requires 2 cores and job class 2 requires 3 cores. The integer solutions within the search space represent the feasible bins. All non-dominated bins exist along the boundary of the polytope since any solution in the polytope not at the boundary will have a point above or to the right of it that is feasible.

We exhaustively enumerate all non-dominated bins. Once a complete set of non-dominated bins is created to represent all assignments of jobs to machines based on expected resource requirements, the machine assignment model decides, for each machine, which bin the machine should emulate. Thus, each machine will be mapped to a single bin, but multiple machines may emulate the same bin.

Algorithm 1 below generates all non-dominated bins. We define K^j, a set of job classes for machine configuration j containing each job class with positive δ^*_{jkl}, and a set b^j containing all possible bins. Given κ^j_i, a job belonging to the i^{th} class in K^j, and

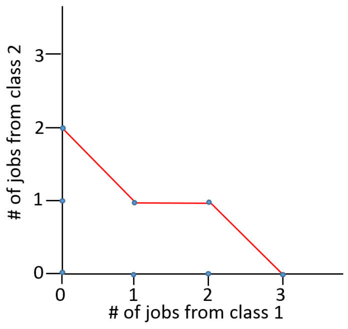


Fig. 5: Feasible bin configurations.

b_y^j, the y^{th} bin for machine configuration j, Algorithm 1 is performed for each machine configuration j. We make use of two functions not defined in the pseudo-code:

- sufficientResource(κ_i^j, b_y^j): Returns true if bin b_y^j has sufficient remaining resources for job κ_i^j.
- mostRecentAdd(b_y^j): Returns the job class that was most recently added to b_y^j.

Algorithm 1 Generation of all non-dominated bins

```
$y \leftarrow 1$
$x \leftarrow 1$
$x^* \leftarrow x$
$nextBin \leftarrow$ **false**
**while** $x \leq |K^j|$ **do**
    **for** $i = x^* \rightarrow |K^j|$ **do**
        **while** sufficientResource($\kappa_i^j$, $b_y^j$) **do**
            $b_y^j \leftarrow b_y^j + \kappa_i^j$
            $nextBin \leftarrow$ **true**
        **end while**
    **end for**
    $x^* \leftarrow$ mostRecentAdd($b_y^j$)
    **if** $nextBin$ **then**
        $b_{y+1}^j \leftarrow b_y^j - \kappa_{x^*}^j$
        $y \leftarrow y + 1$
    **else**
        $b_y^j \leftarrow b_y^j - \kappa_{x^*}^j$
    **end if**
    **if** $b_y^j == \{\}$ **then**
        $x \leftarrow x + 1$
        $x^* \leftarrow x$
    **else**
        $x^* \leftarrow x^* + 1$
    **end if**
**end while**
```

Algorithm 1 is run for each machine configuration j. The algorithm starts by greedily filling the bin with jobs from a class. When no additional jobs from a class can be added, the algorithm will move to the next class of jobs and attempt to continue filling the bin. Once no more jobs from any class are able to fit, the bin is non-dominated. The algorithm then backtracks by removing the last job added and tries to add jobs from other classes to fill the remaining unused resources. This continues until the algorithm has exhaustively searched for all non-dominated bins.

Since the algorithm performs an exhaustive search, solving for all non-dominated bins may take a significant amount of time. If we let L_k represent the maximum number of jobs of class k we can fit onto the machine of interest, then in the worst case, we must consider $\prod_{k\in K} L_k$ bins to account for every potential mix of jobs. We can improve the performance of the algorithm by ordering the classes in decreasing order of resource requirement. Of course, this is made difficult as there are multiple resources. One would have to ascertain the constraining resource on a machine and this may be dependent on which mix of jobs is used.[1]

Although the upper bound on the number of bins is very large, we are able to find all non-dominated bins quickly (i.e., within one second on an Intel Pentium 4 3.00 GHz CPU) because the algorithm only considers job classes with non-zero δ^*_{jkl} values. We generally see a small subset of job classes assigned to a machine configuration. Table 1 in Section 5 illustrates the size of K^j, the number of job classes with non-zero δ^*_{jkl} values for each configuration. When considering four job classes, all but one configuration has one or two job classes with non-zero δ^*_{jkl} values. When running Algorithm 1, the number of bins generated is in the thousands. Without the δ^*_{jkl} values from the allocation LP, we find that there can be on the order of millions of bins.

With the created bins, individual machines are then assigned to emulate one of the bins. To match the δ^*_{jkl} values for the corresponding machine configuration, we must find the contribution that each bin makes to the amount of resources allocated to each job class. We define N_{ijk} as the number of jobs from class k that are present in bin i of machine configuration j. Using the expected resource requirements, we can calculate the amount of resource l on machine j that is used for jobs of class k, denoted $\epsilon_{ijkl} = N_{ijk} r_{kl}$. The machine assignment LP is then

$$\max \quad \lambda \tag{6}$$

$$\text{s.t.} \sum_{j\in M} \Delta_{jkl}\mu_{jk} \geq \lambda \alpha_k r_{kl} \qquad k \in K, l \in R \tag{7}$$

$$\sum_{i\in B_j} \epsilon_{ijkl} x_{ij} = \Delta_{jkl} \qquad j \in M, k \in K, l \in R \tag{8}$$

$$\sum_{i\in B_j} x_{ij} = n_j \qquad j \in M \tag{9}$$

$$x_{ij} \geq 0 \qquad j \in M, i \in B_j \tag{10}$$

[1] It may be beneficial to consider the dominant resource classification of Dominant Resource Fairness when creating such an ordering [8].

Decision Variables

Δ_{jkl}:	Amount of resource l from machine configuration j that is devoted to job class k
x_{ij}:	Total number of machines that are assigned to bins of type i in machine configuration j

Parameters

ϵ_{ijkl}:	Amount of resource l of a machine in machine configuration j assigned to job class k if the machine emulates bin i.
B_j:	Set of bins in machine configuration j

The machine assignment LP will map machines to bins with the goal of maximizing the arrival rate that maintains a stable system. Constraint (7) is the equivalent of constraint (2) of the allocation LP while accounting for discrete machines. The constraint ensures that a sufficient number of resources are available to maintain stability for each class of jobs. Constraint (8) determines the total amount of resource l from machine configuration j assigned to job class k to be the sum of each machine's resource contribution. In order to guarantee that each machine is mapped to a bin type, we use constraint (9). Finally, constraint (10) forces x_{ij} to be non-negative.

Although we wish each machine to be assigned exactly one bin type, such a model requires x_{ij} to be an integer variable and therefore the LP becomes an integer program (IP). We found experimentally that solving the IP model for this problem is not practical given a large set B_j. Therefore, we use an LP that allows the x_{ij} variables to take on fractional values. Upon obtaining a solution to the LP model, we must create an integer solution. The LP solution will have q_j machines of configuration j which are not properly assigned, where q_j can be calculated as

$$q_j = \sum_{i \in B_j} x_{ij} - \lfloor x_{ij} \rfloor.$$

We assign these machines by sorting all non-integer x_{ij} values by their fractionality $(x_{ij} - \lfloor x_{ij} \rfloor)$ in non-increasing order. Ties are broken arbitrarily if there are multiple bins with the same fractional contribution. We then begin to round the first q_j fractional x_{ij} values up and round all other x_{ij} values down for each configuration. This makes the problem tractable at the cost of optimality. However, given the scale of the problem that we study where a configuration can contain thousands of machines, the value of λ^* produced by the LP solution is typically very close to the value produced by the IP solution.

4.3 Dispatching Jobs

In the third and final stage of the scheduling algorithm, a two-level dispatching algorithm is used to assign arriving jobs to machines. The goal of the dispatching algorithm is to assign jobs to machines so that each machine emulates the bin it was assigned to in the second stage. In the first level of the dispatcher, a job is assigned to one of the $|M|$ machine configurations. The decision is guided by the Δ_{jkl} values to ensure that the correct proportion of jobs is assigned to each machine configuration. In the second level of the dispatcher, the job is placed on one of the machines in the configuration to which it was assigned. At the first level, no state information is required to make decisions. However, in the second level, the dispatcher will make use of the exact resource requirements of a job as well as the states of machines to make a decision.

Deciding which machine configuration to assign a job to can be done by revisiting the total amounts of resources each configuration contributes to a job class. We can compare the $\varDelta_{jkl}$ values to create a policy that will closely imitate the machine assignment solution. Given that each job class k has been devoted a total of $\sum_{j=1}^{|M|} \varDelta_{jkl}$ resources of type l, a machine configuration j should serve a proportion

$$\rho_{jk} = \frac{\varDelta_{jkl}}{\sum_{m=1}^{|M|} \varDelta_{mkl}}$$

of the total jobs in class k. The value of ρ_{jk} can be calculated using the $\varDelta_{jkl}$ values from any resource type l. To decide which configuration to assign an arriving job of class k, we use roulette wheel selection. We generate a uniformly distributed random variable, $u = [0, 1]$ and if

$$\sum_{m=0}^{j-1} \rho_{mk} \leq u < \sum_{m=0}^{j} \rho_{mk},$$

then the job is assigned to machine configuration j.

The second step will then dispatch the jobs directly onto machines. Given a solution x_{ij}^* from the machine assignment LP, we create an $n_j \times |K|$ matrix, $\mathbf{A}^j$, with element $\mathbf{A}_{ik}^j$ equal to 1 if the ith machine of j emulates a bin with one or more jobs of class k assigned. $\mathbf{A}^j$ indexes which machines can serve a job of class k.

The dispatcher will attempt to dispatch the job to a machine belonging to the configuration that was assigned from the first step. Machines are ordered arbitrarily and the dispatcher will search over the machines based on the ordering. The first machine found from those with $\mathbf{A}_{ik}^j = 1$ that has the available resources for the job will begin immediate service; this is a first-fit policy that is used by the dispatcher. In the case where no machines are available, the dispatcher will sort all machines, other than the machines belonging to the configuration that the job was initially assigned to, in non-decreasing order of processing times of the job needing assignment. The dispatcher will then search through these machines for immediate processing and if a machine exists with sufficient resources to immediately process the job, it will begin servicing the job. By allowing for the dispatcher to make assignments to machines with $\delta_{jkl}^* = 0$, we enable free resources to be used immediately. One could expect that a system that is not heavily loaded could benefit from the prompt service of jobs arriving to the system even though the assignment is inherently inefficient according to the allocation LP solution. If there exists no machine that can immediately process the job, the job will enter the smallest queue of the machines belonging to the configuration assigned in the first step with $\mathbf{A}_{ik}^j = 1$. Ties are broken randomly. Following such a dispatch policy attempts to schedule jobs immediately whenever possible with a bias towards placing jobs on bins which have been found to be efficient.

Jobs that are waiting in the queue follow a first-come, first-served (FCFS) order. An arriving job will have to wait until all jobs that arrived earlier have at least entered into service before it too can begin processing on the machine. This ensures that some level of fairness is maintained and prevents jobs with smaller resource requests from jumping forward in the queue and possibly starving jobs with large resource requests.

# of machines	Cores	Memory	$\lvert K^j \rvert$
6732	0.50	0.50	4
3863	0.50	0.25	2
1001	0.50	0.75	1
795	1.00	1.00	2
126	0.25	0.25	2
52	0.50	0.12	1
5	0.50	0.03	1
5	0.50	0.97	2
3	1.00	0.50	2
1	1.00	0.06	1

Table 1: Machine configuration details.

We use this ordering because it is often the default scheduling sequence used in practice for frameworks that run jobs in a data center environment, such as Hadoop [1].

By dispatching jobs using the proposed algorithm, the requirement of system state information is often reduced to a subset of machines that a job is potentially assigned to. Further, keeping track of the detailed schedule on each machine is not necessary for scheduling decisions since the only information used is whether a machine currently has sufficient resources, which job is next to be scheduled in the queue, and the size of the queue.

5 Experimental Results

We test our algorithm using cluster workload trace data provided by Google.[2] This data represents the workload for one of Google's compute clusters over the one month period of May 2011. The data captured in the trace workload provides information on the machines in the system as well as the jobs that arrive, their submission times, their resource requests, and their durations, which can be inferred from finding how long a job is active. However, because we calculate the processing time of a job based on the actual processing time realized in the workload traces, it is unknown to us how processing times may have differed if a job was processed on a different machine. Therefore, we assume that processing times are independent of machine configuration. In-depth analysis on the workload has been previously done [21]; we will be using the data as input for our scheduling algorithm to simulate its performance over the one month period.

Although the information provided is extensive, we limit what we use for our experiments. We do not consider failures of machines or jobs. Resubmitted jobs due to failures are considered to be new, unrelated jobs. Machine configurations change over time due to failures, the acquisition of new servers, or the decommissioning of old ones, but we will only use the initial set of machines and keep that constant over the whole month. Furthermore, system micro-architecture is provided for each machine. Some jobs are limited in which types of architecture they can be paired with and how they

[2] The data can be found at https://code.google.com/p/googleclusterdata/.

Job class	1	2	3	4
Avg. Time (h)	0.03	0.04	0.04	0.03
Avg. Cores	0.02	0.02	0.07	0.20
Avg. Mem.	0.01	0.03	0.03	0.06
Proportion of Total Jobs	0.23	0.46	0.30	0.01

Table 2: Job class details.

interact with these architectures, but we ignore this limitation for our scheduling experiments. It is easy to extend the LoTES algorithm to account for system architecture by setting $\mu_{jk} = 0$ whenever a job cannot be processed on a particular architecture. The focus for our work is on the efficient allocation of server resources to job classes and so we abstract the trace data to look only at resource requests and job durations.

The cluster of interest has 10 machine configurations (we use the configurations provided from the Google workload trace data) as presented in Table 1. Each configuration is defined strictly by its resource capacity and the number of identical machines with that resource profile. The resource capacities are normalized relative to the configuration with the most resources. Therefore, the job resource requests are also provided after being normalized to the maximum capacity of machines.

5.1 Class Clustering

The Google data does not define job classes and so in order for us to use the data to test our LoTES algorithm, we must first cluster jobs into classes. We follow Mishra et al. [18] by using k-means clustering to create job classes. We make use of Lloyd's algorithm [15] to create the different clusters. To limit the amount of information that LoTES is using in comparison to our benchmark algorithm, we only use the jobs from the first day to define the job classes for the month. These classes are assumed to be fixed for the entire month. Due to this assumption and because the Greedy policy does not use class information, any inaccuracies introduced by making clusters based on the first day will only make LoTES worse when we compare the two algorithms.

Clustering showed us that four classes were sufficient for representing most jobs. Increasing the number of classes led to less than 0.01% of jobs being allocated to the new classes and therefore, we use only four classes in our experiments. The different job classes are presented in Table 2.

5.2 Benchmark Algorithm: A Greedy Dispatch Policy

To illustrate the performance of the LoTES algorithm, we propose a Greedy dispatch policy as a benchmark. We chose to compare LoTES against the Greedy dispatch policy because it is a natural heuristic, which aims to quickly process jobs. The dispatch policy, like the LoTES algorithm, attempts to schedule jobs onto available machines immediately if possible. If a machine is found that can immediately process a job, the dispatch policy will make that assignment. In the case where no machines are available for immediate processing, the policy will choose the machine with the shortest queue of waiting jobs. Ties are broken randomly. However, an assignment cannot be made if the requested resources of a job by themselves exceed the capacity of a machine. If a queue forms, jobs will be processed in FCFS order.

The Greedy dispatch policy is similar to the LoTES algorithm. The key difference in the two approaches is that the LoTES algorithm restricts the set of machines it considers to the set of machines found from solving the higher level allocation problems in the first two stages. By comparing against the Greedy policy, we can test how effective LoTES is and how useful the proper machine-job mapping is to system performance.

5.3 Implementation Challenges

In our experiments, we have not directly considered the time it takes for the scheduler to make dispatching decisions. As such, as soon as a job arrives to the system, the scheduler will immediately assign it to a machine. In practice, decisions are not instantaneous and depending on the amount of information needed by the scheduler and the complexity of the scheduling algorithm, the delay may be an issue. For every new job arrival, the scheduler requires state information of one or more machines. The state of the machine must provide the currently available resources and the size of the queue. As the system becomes busier, the scheduler may have to obtain state information for all machines in the data center. Thus, scaling may be problematic as the algorithms may have to potentially search over a very large number of machines. However, in heavily loaded systems where there are delays before a job can start processing, the scheduling overhead will not adversely affect system performance so long as the overhead is less than the waiting time delays. An additional issue may be present that could reduce performance of the scheduler at heavy loads. The scheduler creates additional load on the network connections within the data center itself. This may need to be accounted for if the network connections become sufficiently congested.

Note, however, that the dispatching overhead of LoTES is no worse than that of the Greedy policy. The LoTES algorithm benefits from the restricted set of machines that it considers when making scheduling decisions, but that does not guarantee that LoTES would not also end up obtaining state information on every machine when the system is heavily loaded. Therefore, a system manager for a very large data center must take into account the overhead required to obtain machine state information. There is work showing the benefits of only sampling state information from a limited set of machines to make a scheduling decision [10]. If the overhead of obtaining too much state information is problematic, we suggest that one can further limit the number of machines to be considered once a configuration has already been chosen. Such a scheduler could decide which configuration to send an arriving job to and then sample N machines randomly from the chosen configuration, where $N \in [1, n_j]$. Restricting the scheduler to only these N sampled machines, the scheduler can dispatch jobs following the same rules as LoTES. This allows the mappings from the offline stages of LoTES to still be used, but with substantially less overhead for the online portion.

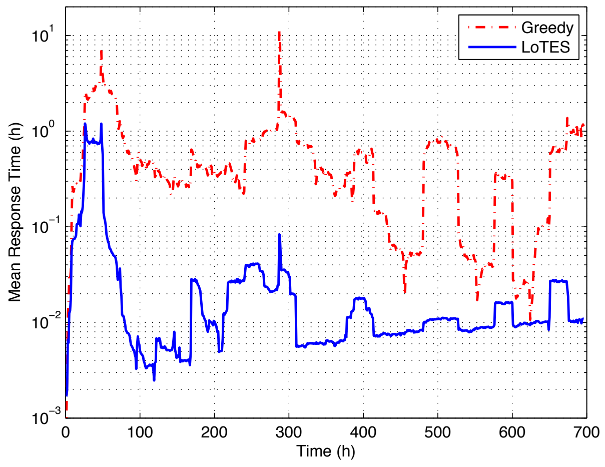


Fig. 6: Response Time Comparison.

5.4 Simulation Results: Workload Trace Data

We simulate the LoTES algorithm and the Greedy dispatch policy using the workload traces from Google. We created an event-based simulator in C++ to emulate a data center with the workload data used as input to our system. The LP models are solved using IBM ILOG CPLEX 12.5. We run our tests on an Intel Pentium 4 CPU 3.00 GHz, 1 GB of main memory, running Red Hat 3.4-6-3. Because the LP models are solved offline prior to the arrival of jobs, the solutions to the first two stages are not time-sensitive. Regardless, the total time required to obtain solutions to both LP models and to generate bins requires less than one minute of computation time. This level of computational effort means that it is realistic to re-solve these two stages periodically, perhaps multiple times a day, if the job classes or machine configurations change due, for example, to non-stationary distributions. We leave this for future work.

Figure 6 presents the performance of the system over the one month period. The graph provides the mean response time of jobs over every one-hour long interval. We include a job's response time in the mean response time calculation in the interval in which the job begins processing. We see that the LoTES algorithm greatly outperforms the Greedy policy. On average, the Greedy policy has response times an order of magnitude longer (15-20 minutes) than the response times of the LoTES algorithm (1-2 minutes). The difference on average shows the strong performance of LoTES, however, a more interesting result is the performance difference when the system becomes heavily loaded. During the one-month period, the Greedy policy has two large spikes in response times that occur where jobs must wait for close to 10 hours around the 70 hour and 280 hour time points. During both occurrences, the LoTES algorithm produces schedules with response times on the order of 1 hour long in the first occurrence, and 10 minutes in the second occurrence.

Figures 7 and 8 provide the core and memory utilization of the machines over time. At the end of each hour, we record the instantaneous utilization of resources over all machines and graph those results. We observe that curves are typically very close to

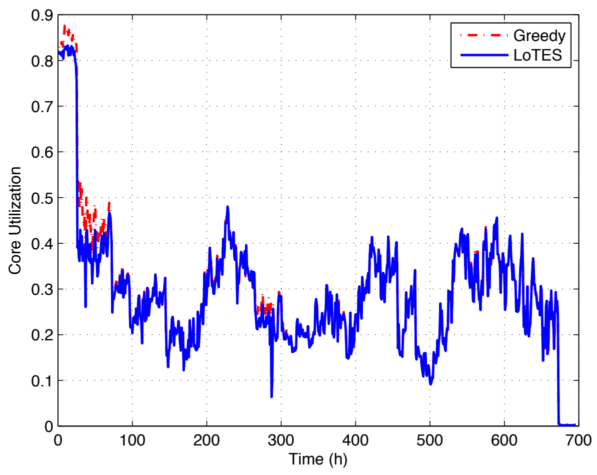


Fig. 7: Processing Core Utilization Comparison.

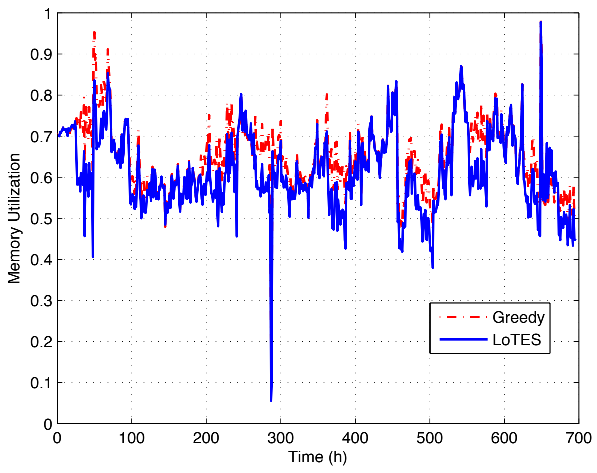


Fig. 8: Memory Utilization Comparison.

each other, but at certain time points, the Greedy policy has higher utilization. We believe the increased utilization is due to the build up of jobs in queue for the Greedy policy. With a large queue, as soon as jobs are completed, the available space is filled again with a waiting job. Since LoTES is doing a better job of increasing throughput in the short term through efficient allocation, queues do not form as often. However, over the long term, as this is an open system and we assume that no jobs are abandoned, the long-run throughput of both systems will be the same and therefore long-run resource utilizations are also the same. As a result, LoTES is doing a better job at smoothing the utilization curves.

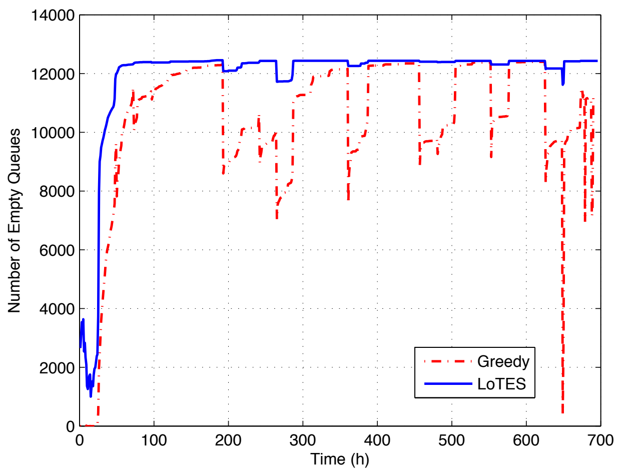


Fig. 9: Empty Queue Comparison.

Figure 9 plots the number of empty queues for both schedulers. Of the 12,583 machines present in the system, we graph the number of machines that have an empty queue during each hour of the simulation. Although the queue length at a machine may change during the hour long period, we only record the state of the queue at the end of the hour. We see that very quickly, the LoTES algorithm is able to keep the queues of all machines relatively empty. However, the Greedy policy often has a large number of machines with a queue. This queue formation leads to the higher resource utilization and the increased response times.

6 Conclusion and Future Work

In this work, we developed a scheduling algorithm that creates a mapping between jobs and machines based on their resource profiles to improve the response time of the system. The algorithm consists of three stages where a fluid representation and queueing model are used at the first stage to fractionally allocate job classes to machine configurations. The second stage then solves a combinatorial problem to generate possible assignments of jobs on machines. An LP model is developed to maximize system capacity by choosing which of the generated sets of jobs that each machine should aim to emulate. The final stage is an online dispatching policy that uses the solution from the second stage to decide on the machine to assign to each incoming job. Our algorithm was tested on Google workload trace data and was found to reduce response times by up to an order of magnitude when compared to a benchmark dispatch policy. This improvement in performance is also computationally cheaper than the benchmark policy during the online scheduling phase since the proposed algorithm often requires state information for fewer machines when making assignment decisions.

The data center scheduling problem is very rich from the scheduling perspective and can be expanded in many different ways. Our algorithm assumes stationary arrivals over the entire duration of the scheduling horizon. However, the real system is not stationary and the arrival rate of each job class may vary over time. Furthermore, the actual job classes themselves may change over time as resource requirements may not always be clustered in the same manner. As noted above, the offline phase is sufficiently fast (about one minute of CPU time) that it could be run multiple times per day as the system and load characteristics change. Beyond this we plan to extend the LoTES algorithm to more accurately represent dynamic job classes. This would allow the LoTES algorithm to learn and predict the expected job population and make scheduling decisions with these predictions in mind. Not only do we wish to be able to adjust our algorithm to a changing environment, but we also wish to extend our algorithm to be able to more intelligently handle situations where there is high variance in the mix of job classes in the environment. The high variance will lead to system realizations that differ significantly from the bins created in the second stage of the LoTES algorithm.

We also plan to study the effects of errors in job resource requests. We used the amount of requested resources of a job as the amount of resource used over the entire duration of the job. In reality, most jobs may end up using less or more resources than requested due to the fact that users may under or overestimate their resource requirements. In addition, the utilization of a resource may change over the duration of the job itself. We plan to incorporate these uncertainties regarding resource usage to improve system utilization. This adds difficulty to the problem because instead of creating a schedule where we know the exact amount of requested resources once a job arrives, we only have an estimate of the requests and must ensure that a machine is not underutilized or oversubscribed.

Acknowledgment

This work was made possible in part due to a Google Research Award and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Apache Hadoop. http://hadoop.apache.org
2. Al-Azzoni, I., Down, D.G.: Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems. IEEE Transactions on Parallel and Distributed Systems **19**(12), 1671–1682 (2008)
3. Andradóttir, S., Ayhan, H., Down, D.G.: Dynamic server allocation for queueing networks with flexible servers. Operations Research **51**(6), 952–968 (2003)
4. Berral, J.L., Goiri, Í., Nou, R., Julià, F., Guitart, J., Gavaldà, R., Torres, J.: Towards energy-aware scheduling in data centers using machine learning. In: Proceedings of the 1st International Conference on energy-Efficient Computing and Networking, pp. 215–224. ACM (2010)
5. Chang, R.S., Lin, C.Y., Lin, C.F.: An adaptive scoring job scheduling algorithm for grid computing. Information Sciences **207**, 79–89 (2012)
6. Dai, J.G., Meyn, S.P.: Stability and convergence of moments for multiclass queueing networks via fluid limit models. IEEE Transactions on Automatic Control **40**(11), 1889–1904 (1995)
7. Gandhi, A., Harchol-Balter, M., Kozuch, M.A.: Are sleep states effective in data centers? In: International Green Computing Conference (IGCC), pp. 1–10. IEEE (2012)
8. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI, vol. 11, pp. 24–24 (2011)

9. Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., Akella, A.: Multi-resource packing for cluster schedulers. In: Proceedings of the 2014 ACM conference on SIGCOMM, pp. 455–466. ACM (2014)
10. He, Y.T., Down, D.G.: Limited choice and locality considerations for load balancing. Performance Evaluation **65**(9), 670–687 (2008)
11. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 261–276. ACM (2009)
12. Kim, J.K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T.D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., et al.: Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. Journal of Parallel and Distributed Computing **67**(2), 154–169 (2007)
13. Le, K., Bianchini, R., Zhang, J., Jaluria, Y., Meng, J., Nguyen, T.D.: Reducing electricity cost through virtual machine placement in high performance computing clouds. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 22. ACM (2011)
14. Liu, Z., Lin, M., Wierman, A., Low, S.H., Andrew, L.L.: Greening geographical load balancing. In: Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp. 233–244. ACM (2011)
15. Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information Theory **28**(2), 129–137 (1982)
16. Maguluri, S.T., Srikant, R., Ying, L.: Heavy traffic optimal resource allocation algorithms for cloud computing clusters. In: Proceedings of the 24th International Teletraffic Congress, p. 25. International Teletraffic Congress (2012)
17. Maguluri, S.T., Srikant, R., Ying, L.: Stochastic models of load balancing and scheduling in cloud computing clusters. In: Proceedings IEEE INFOCOM, pp. 702–710. IEEE (2012)
18. Mishra, A., Hellerstein, J., Cirne, W., Das, C.: Towards characterizing cloud backend workloads: insights from Google compute clusters. ACM SIGMETRICS Performance Evaluation Review **37**(4), 34–41 (2010)
19. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 69–84. ACM (2013)
20. Rasooli, A., Down, D.G.: COSHH: A classification and optimization based scheduler for heterogeneous hadoop systems. Future Generation Computer Systems **36**, 1–15 (2014)
21. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing, pp. 1–13. ACM (2012)
22. Tang, Q., Gupta, S.K., Varsamopoulos, G.: Thermal-aware task scheduling for data centers through minimizing heat recirculation. In: IEEE International Conference on Cluster Computing, pp. 129–138. IEEE (2007)
23. Wang, L., Von Laszewski, G., Dayal, J., He, X., Younge, A.J., Furlani, T.R.: Towards thermal aware workload scheduling in a data center. In: Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on, pp. 116–122. IEEE (2009)
24. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on Computer systems, pp. 265–278. ACM (2010)