Journal of Scheduling: MISTA Special Issue

Multi-Stage Resource-Aware Scheduling for Data Centers with Heterogeneous Servers

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

Abstract This paper presents a three-stage algorithm for resource-aware scheduling of computational jobs in a large-scale heterogeneous data center. The algorithm aims to allocate job classes to machine configurations to attain an efficient mapping between job resource request profiles and machine resource capacity profiles. 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 solve a shorter-term, more accurate representation of the problem using the first stage, long-term solution for heuristic guidance. In the second stage, jobs and machines are discretized. A linear programming model is used to obtain a solution to the discrete problem that maximizes the system capacity given a restriction on the job class and machine configuration pairings based on the solution of the first stage. The final stage is a scheduling policy that uses the solution from the second stage to guide the dispatching of arriving jobs to machines. We present experimental results of our algorithm on both Google workload trace data and generated data and show that it outperforms existing schedulers. These results illustrate the importance of considering heterogeneity of both job and machine configuration profiles in making effective scheduling decisions.

[+] Department of Mechanical and Industrial Engineering,
University of Toronto
E-mail: {tran, meghanap, 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

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. Providers of these services employ large data centers, but as the demand for these services increases, performance can degrade if the data centers are not sufficiently large or are being utilized inefficiently. Due to the capital required for the machines, many data centers are not purchased as a whole at one time, but rather built incrementally, adding machines in batches as demand increases. Data center managers may choose machines based on the price-performance trade-off that is economically viable and favorable at the time [23]. Therefore, it is not uncommon to see data centers comprised of tens of thousands of machines, which are divided into 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. Such 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 machine configurations so that efficient routing of jobs to machines can be made to improve response times to end users. 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 applied to an abstracted representation of the problem, based on pooled resources and jobs. In each successive stage, a finer system model is used, such that in the third stage we dispatch jobs to machines. Our experiments are based on both job traces from one of Google's compute clusters [20] and carefully generated instances that test behaviour as relevant independent variables are varied. We show that our algorithm outperforms a natural greedy policy that attempts to minimize the response time of each arrival and the Tetris scheduler [7], a dispatching policy that adapts heuristics for the multi-dimensional bin packing problem to data center scheduling.[1]

The contributions of this paper are:

- A hybrid queueing theoretic and combinatorial optimization scheduling algorithm for a data center that performs significantly better than existing techniques tested.
- An extension to the allocation linear programming (LP) model [2] used for distributed computing [1] to a data center that has machines with multi-capacity resources.
- An empirical study of our scheduling algorithm on both real workload trace data and randomly generated data.

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 and suggests directions for future work.

2 Problem Definition

The data center of interest is comprised of on the order of tens of thousands of independent servers (also referred to as machines). These machines are not all identical; the machine population is divided into different configurations denoted by the set M. Machines belonging to the same configuration are identical in all aspects.

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. A machine of configuration $j \in M$ has c_{jl} amount of resource $l \in R$,

[1] Earlier work on our algorithm, appearing at the Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA) 2015 presented a comparison only to the Greedy policy. We have extended the paper by improving our algorithm, including a comparison to the Tetris scheduler, and significantly expanding the experimentation.

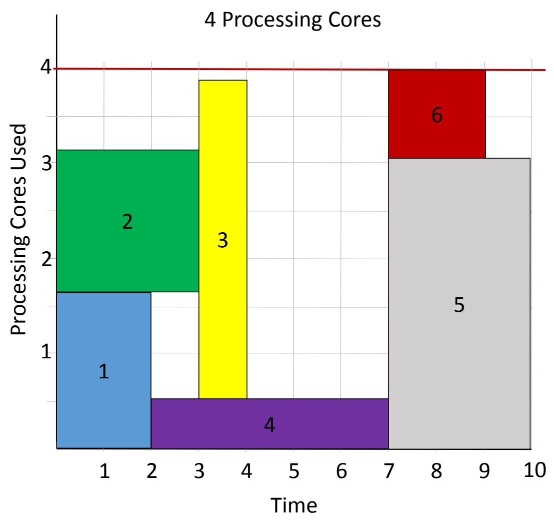

Fig. 1 Processing cores resource consumption profiles

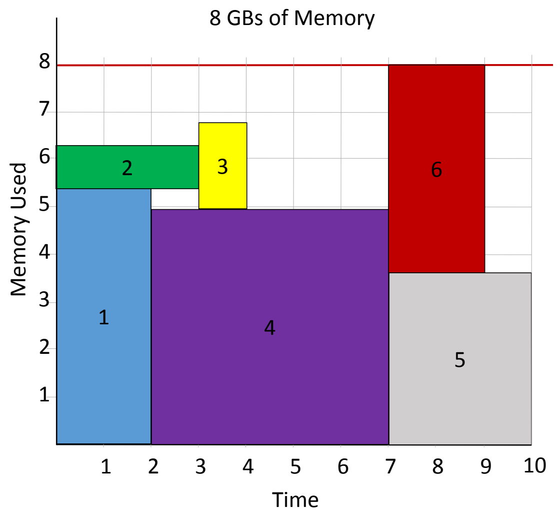

Fig. 2 Memory resource consumption profiles

which defines the machine's resource profile. Within a configuration j there are n_j identical machines.

In our problem, jobs must be assigned to the machines with the goal of minimizing the mean response time of the system. Jobs arrive to the data center dynamically over time with the intervals between arrivals being 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 $k \in K$ is α_k. We denote the expected amount of resource of type l required by a job of class k as r_{kl}. The resources required by a job define its resource profile, which can be differ-

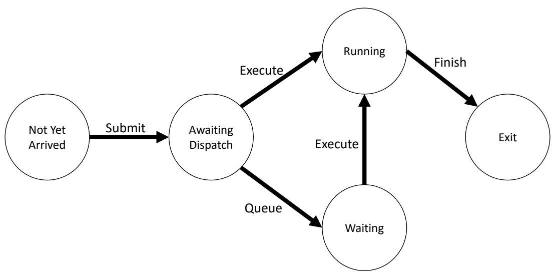

Fig. 3 Stages of job lifetime.

ent from the resource profile of the job class as the latter is an estimate of a job's actual profile. 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 the amount of resource. The machine has four processing cores and eight 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 where jobs can queue until they begin processing. Figure 3 illustrates the 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 can either be scheduled onto a machine if there are sufficient resources or it can enter the queue and await execution. After being processed, the job will exit the data center.

The key challenge in the allocation of jobs to machines is that the resource usage is unlikely to exactly match the resource capacity. As a consequence, small amounts of each resource will be unused. This phenomenon is called *resource fragmentation* because while there may be enough resources to serve another job, they are spread across different machines. For example, if a configuration has 30 machines with eight cores available on each machine and each job assigned to the configuration requires exactly three cores, the pooled machine can process 80 jobs in parallel on its 240 processors. In reality, of course, only two jobs can be placed on each machine and so 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. Thus, producing high quality schedules is a difficult task when faced with resource fragmentation under dynamic job arrivals.

3 Related Work

Scheduling in data centers has received significant attention in the past decade. Mann [19] presents many problem contexts and characteristics as the literature has focused on different aspects of the problem. Unfortunately, as Mann points out, the approaches are mostly incomparable due to subtle differences in the problem models. For example, some works consider cost saving through decreased energy consumption from lowering thermal levels [25,28], powering down machines [3, 5], or geographical load balancing [14,15]. These works often attempt to minimize costs or energy consumption while maintaining some guarantees on response time and throughput. Other works are concerned with balancing energy costs, service level agreement performance, and achieving a level of reliability [8,9,24].

The literature on schedulers for distributed computing clusters has focused heavily on *fairness* and *locality* [11,21,29]. Optimizing these performance metrics leads to equal access to resources for different users and the improvement of performance by assigning tasks close to the location of stored data to reduce data transfer traffic. Locality of data has been found to be crucial for performance in systems such as MapReduce, Hadoop, and Dryad when bandwidth capacity is limited [29]. Our work does not consider data transfer or equal access for different users as the problem we consider focuses on the heterogeneity of machines with regards to resource capacity. The characteristic of resource heterogeneity and fragmentation that we study is an already considerable scheduling challenge. We hope to incorporate locality and fairness into our model as future work.

The literature on machine heterogeneity has some key differences from our model. One area of research considers heterogeneity in the form of processing time and not resource usage and capacity [1,13,22]. Here, the processing time of a job is dependent on the machine that processes the job. Without a model of resource usage, fragmentation cannot be reasoned about, but efficient allocation of jobs to resources can still be an important decision. Kim et al. [13] study dynamic mapping of jobs to machines with varying priorities and soft deadlines. They find that two scheduling heuristics stand out as the best performers: *Max-Max* and *Slack Sufferage*. In the former, 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 processing time. *Slack Sufferage* chooses job mappings based on which jobs suffer most if not scheduled onto their "best" machines. Al-Azzoni and Down [1] schedule jobs to machines using a linear program (LP) to efficiently pair job classes to machines based on their expected processing times. The solution of the LP maximizes the system capacity and guides the scheduling rules to reduce the long-run average number of jobs in the system. Further, they are able to show that their heuristic policy is guaranteed to be stable if the system can be stabilized. Another study that considers processing time as a source of resource heterogeneity extends the allocation LP model to address a Hadoop framework [22]. The authors compare their work against the default scheduler used in Hadoop and the *Fair-Sharing* algorithm and demonstrate that their algorithm greatly reduces the mean response time, while maintaining competitive levels of fairness with Fair-Sharing. These studies illustrate the importance of scheduling with processing time heterogeneity in mind. While the focus of our work is resource capacity heterogeneity, we are able to demonstrate strong performance in experiments that also include processing time heterogeneity (see Section 5.4.3).

Some work that studies resource usage and capacity as the source of heterogeneity in a system makes use of a limited set of virtual machines with pre-defined resource requirements to simplify the issue of resource fragmentation. Maguluri et al. [18] examine a cloud computing cluster where virtual machines are to be scheduled onto servers. There are three different types of virtual machines: *Standard*, *High-Memory*, and *High-CPU*, each with specified resource requirements common to all virtual machines of a single type. 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 considers the predefined virtual machine combinations on servers and is shown to be throughput-optimal. Maguluri et al. later extended their work to a queue-length optimal algorithm for the same problem in the heavy traffic regime [17]. They propose a routing algorithm that assigns jobs to servers with the shortest queue (similar to our Greedy algorithm presented in Section 5.1) and a mix of virtual machines to assign to a server based on the same reasoning proposed for their throughput optimal algorithm. Since the virtual machines have predetermined resource requirements, it is known exactly how virtual machine types will fit on a server without having to reason online about each assignment individually. This difference from our problem means it is possible to obtain performance guarantees for the scheduling policies as one can accurately account for the resource utilization of the virtual machines. However, the performance guarantees are only with respect to virtual machines which represent upper bounds on the true resource usage. Fragmentation will occur *across* virtual machines when a job does not utilize all the resource in the virtual machine it is assigned to.

Ghodsi et al. [6] examine a system where fragmentation does occur, but they do not try to optimize job allocation to improve response time or resource utilization. Their focus is solely on fairness of resource allocation through the use of a greedy algorithm called Dominant Resource Fairness (DRF). A dominant resource is defined as the one for which the user has the highest requirement normalized by the maximum resource capacity over all configurations. 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 with the goal that the proportion of dominant resources for each user is fair following Jain's Fairness Index [12]. Note that this approach compares resources of different types as the consideration is based on a user's dominant resource.

The work closest to ours is the Tetris scheduler [7]. Tetris considers resource fragmentation and response time as a performance metric. In addition, fairness is also integrated into their model. The Tetris scheduler considers a linear combination of two scoring functions: best fit and least remaining work first. The first score favours large jobs, while the second favours small jobs. Tetris combines these two scores for each job and then chooses the next job to process based on the job with the highest score. Tetris is compared against DRF and it is demonstrated that focusing on fairness alone can lead to poor performance, while efficient resource allocation can be important. We directly compare our scheduling algorithm to Tetris in Section 5 as it is the most suitable model with similar problem characteristics and performance metrics.

4 Data Center Scheduling

The problem we address requires the assignment of dynamically arriving jobs to machines. Each job has a resource requirement profile that is known once the job has arrived to the system. Machines in our data center each belong to one machine configuration and each configuration has many identical machines with the same resource capacities. The performance metric of interest is the minimization of the system's average job response time.

We propose Long Term Evaluation Scheduling (LoTES), 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, using an allocation LP to represent the queueing system as a fluid model where incoming jobs can be considered in the aggregate as a continuous flow [2]. We extend the LP model from the literature to account for multiple resources in our data center system. The LP is used to find an efficient pairing of machine configurations to job classes. The efficient allocations are then used to restrict the pairings that are considered in the second stage where a machine assignment LP model is used to assign specific machines to serve job classes. In the final stage, jobs are dispatched to machines dynamically as they arrive to the system with the goal of mimicking the assignments from the second stage.

4.1 Stage 1: Allocation of Machine Configurations

Andradóttir et al.'s [2] 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 [4] that can informally be understood as implying that the expected queue lengths in the system remain bounded over time. It is important to ensure that a system is stable, otherwise performance will quickly deteriorate. Although stability alone is not sufficient to ensure that the system will have short response times, finding the maximum arrival rate for a data center, along with the allocation of resources to obtain system stability with that rate, will provide efficient resource usage to improve throughput.

We modify the allocation LP to accommodate $|R|$ resources. Additionally, the large number of machines is reduced by combining 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, treating the amount of incoming work of all jobs (a product of the processing time and resource requirements) as a continuous fluid to be allocated to these super-machines in such a way as to maximize the amount of work that can be sustained. Thus, the allocation LP is a relaxation of the actual system where jobs must be treated as discrete, indivisible tasks rather than continuous amounts of work and machines do not share resources in an aggregated manner. These two relaxations together allow the allocation LP the freedom to divide a job across multiple machines.

As an example assume jobs are processed at a rate of one job per minute on a machine and there exist two machines. There is only a single resource with the machines each having a capacity of five and jobs requiring three units of the resource. In practice, only a single job can be processed on each machine, so the maximum number of jobs that this system can handle is two jobs per minute. If more than two jobs arrive each minute, the system will acquire a queue that will continue to grow. The relaxation will treat the machines as a super-machine that has a capacity of 10 and furthermore, jobs are divisible such that a machine can process a job while it has fewer resources than required, but at a slower rate. Then it is possible to fit $\frac{10}{3}$ jobs on the super-machine at any time and so the relaxed system can handle $\frac{10}{3}$ job arrivals per minute.

The extended allocation LP is:

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

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

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

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

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

The decision variable, λ, denotes the arrival rate of jobs to the system and the objective is to maximize that rate, while maintaining stability. The LP determines δ_{jkl}, the fractional amount of resource l that super-machine j devotes to job class k. Constraint (2) guarantees that sufficient resources are allocated for the

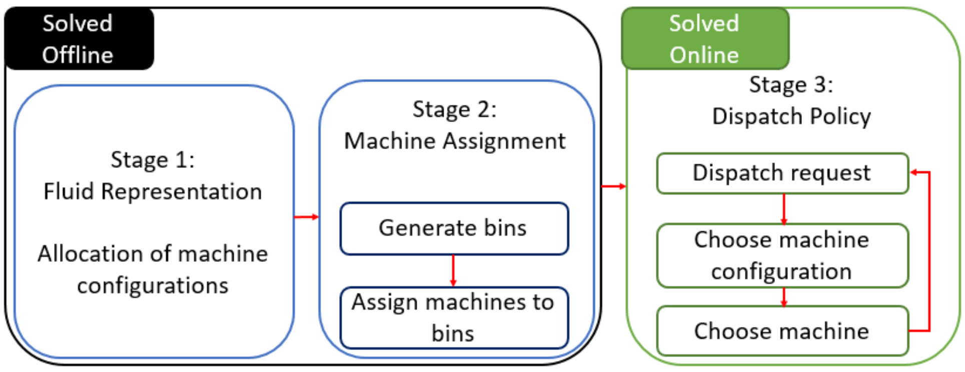
Solved Offline
Stage 1: Fluid Representation
Allocation of machine configurations
Stage 2: Machine Assignment
Generate bins
Assign machines to bins
Solved Online
Stage 3: Dispatch Policy
Dispatch request
Choose machine configuration
Choose machine

Fig. 4 LoTES Algorithm.

expected requirements of each class. Constraint (3) ensures that the resource profiles of the job classes 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 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 to efficiently allocate jobs to machine configurations. The second stage of LoTES will make use the allocation LP solution to guide its search in an attempt to mimic these efficient allocations while accounting for the discrete jobs and machines.

4.1.1 Rationale for the Fluid Model

The first stage of our algorithm provides efficient matchings between job classes and machine configurations for the latter two stages. Although the problem solved for this stage is a relaxation, it captures the long-term behaviour of the system.

Our hypothesis is that we need to reason about both the long-term stochastic behaviour of the system and its short-term combinatorial aspects. As optimal solutions to the combined problem are beyond existing optimization techniques, we choose to optimally solve a relaxation that focuses on the long-term performance and then use that solution to guide reasoning on the combinatorial components.

The allocation LP builds upon the strong analytical results from the queueing theory literature that are able to deduce tight upper bounds on the achievable capacity and prescribe dispatching rules to achieve the calculated bounds with an arbitrarily small approximation [1,2]. What distinguishes our allocation LP from that of previous work is the inclusion of multiple resources with capacity. This addition leads to fragmentation, which results in the loss of the bound guarantee and in the need for combinatorial reasoning. However, even without tight bounds on the capacity of a network, by taking into account the allocation LP results, the later stages of LoTES incorporate information about the long-term behaviour of the system. Typically, such information is unavailable to combinatorial algorithms [27].

4.2 Stage 2: Machine Assignment

In the second stage if the algorithm, we use the job-class-to-machine-configuration results from the allocation LP to guide the choice of a configuration 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 configurations of jobs (which we call "bins") that result in tightly packed machines and then deciding which bin each machine will emulate. As this stage is also offline, we continue to use the expected resource requirements for each job class.

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, the expected resource requests of jobs and the capacities of the machines, the machine assignment algorithm will first create job bins. A bin is any multi-set of job classes that together do not exceed the capacity of the machine (in expectation). A *non-dominated bin* is one that 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 seven. There are two job classes, job class 1 requires two cores and job class 2 requires three cores. The integer solutions represent

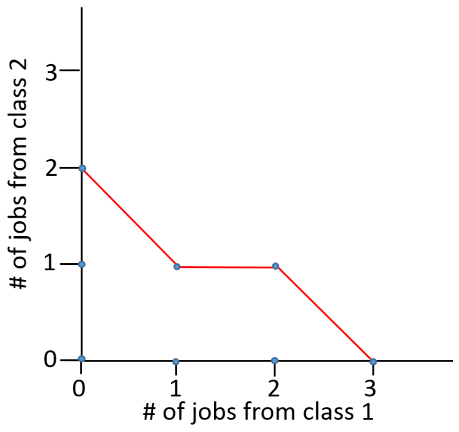

Fig. 5 Feasible bin configurations.

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 that is feasible.

We exhaustively enumerate all non-dominated bins. The machine assignment model then decides which bin each 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 b^j_y, 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^j_i, b^j_y): Returns true if bin b^j_y has sufficient remaining resources for job κ^j_i.
- mostRecentAdd(b^j_y): Returns the job class that was most recently added to b^j_y.

The algorithm starts by greedily filling the bin with jobs from a class. When no additional jobs from that 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 searches for another non-dominated bin by removing the last job added and trying 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 that 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.[2]

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($\kappa^j_i$, $b^j_y$) do
            $b^j_y \leftarrow b^j_y + \kappa^j_i$
            $nextBin \leftarrow TRUE$
        end while
    end for
    $x^* \leftarrow$ mostRecentAdd($b^j_y$)
    if $nextBin$ then
        $b^j_{y+1} \leftarrow b^j_y - \kappa^j_{x^*}$
        $y \leftarrow y + 1$
    else
        $b^j_y \leftarrow b^j_y - \kappa^j_{x^*}$
    end if
    if $b^j_y == \{\}$ then
        $x \leftarrow x + 1$
        $x^* \leftarrow x$
    else
        $x^* \leftarrow x^* + 1$
    end if
end while
```

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 1 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, there can be 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

[2] It may be beneficial to consider the dominant resource classification of Dominant Resource Fairness when creating such an ordering [6].

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 $\epsilon_{ijkl} = N_{ijk} r_{kl}$. We then solve a second LP to assign machines as follows:

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

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

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

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

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

Here, the decision variables are λ, the arrival rate of jobs, Δ_{jkl}, the amount of resource l from machine configuration j that is devoted to job class k, and x_{ij}, the total number of machines from configuration j that are assigned to bins of type i. 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. Here, ϵ_{ijkl} is the amount of resource l of a machine in configuration j that is assigned to job class k if the machine emulates bin i and B_j is the set of bins in configuration j. 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). However, 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, where ties are broken arbitrarily if there are multiple bins with the same fractional contribution. We then round the first q_j fractional x_{ij} values up and round all other x_{ij} values down for each configuration.

The rounding procedure is guaranteed to generate a feasible solution for the machine assignment LP. Constraint (9) naturally follows due to the way that rounding is performed selectively to round up the correct number of fractional x_{ij} variables and round down the remainder. Based on these updated integer x_{ij} values, Δ_{jkl} will be calculated accordingly in Constraint (8), which in turn dictates the maximum λ value for Constraint (7).

4.2.1 Rationale for the Machine Assignment Problem

The second stage of our algorithm reasons about the combinatorial aspects of the system. Unlike in the first stage that uses a fluid relaxation to ensure that the resulting model is tractable, the machine assignment LP restricts decisions based on the allocation LP solution and considers a combinatorial optimization problem that is tractable via relaxation of the IP.

The generated bins make use of expected resource requirements as this is the most accurate way to represent the resource usage of jobs on machines without using stochastic models. Although stochastic models can potentially provide a more accurate representation, it is not clear how to model such a system, given that decisions in the third stage will dictate the correct model to use, or how to solve the resulting stochastic model.

The bins generated are restricted by the δ^*_{jkl} values obtained by the allocation LP. We chose to restrict the system as such because the δ^*_{jkl} solution represents what is, for the relaxed problem, an efficient matching and considerably reduces the number of possible bins based on this efficient matching. The bin generating problem is similar to the multi-dimensional knapsack problem with an exponentially large search space, representing the number of unique bins that can be generated.

The second step, the machine assignment LP, is an extension of the allocation LP that combines aspects of the first stage LP along with discretized bins and machines. However, the machine assignment LP does not exactly model the system with discrete machines since the assignment allows for a fractional number of machines to be assigned to a bin. We chose this representation because the LP problem is tractable and does not lead to significantly worse solutions. We round the LP solution to integer values. However, these variables represent the number of machines assigned to a bin. These values tend to be in the hundreds or thousands while the error due to rounding is, of course, less than 0.5. Therefore, the use of the LP instead of the IP does not significantly impact the quality of the solution (i.e., we observed a reduction in the solution quality of less than 0.001% due to rounding). Furthermore, since the model presented thus far is an approximation of the system rather than a perfectly accurate representation, optimizing for such small differences is unlikely to provide meaningful performance improvements.

4.3 Stage 3: Dispatching Policy

In the third and final stage of the scheduling algorithm, jobs are dispatched to machines. There are two events that change the system state such that a scheduling decision can be made. The first event is a job arrival where the scheduler can assign the arriving job to a machine. However, it may be that machines do not have sufficient resources and so the job must enter a queue and wait until it can be processed by a machine. The second event is the completion of a job. Once a job has finished processing, resources on the machine become available again and if there are jobs in queue that can fit on the machine, the scheduler can have the machine begin processing the job. However, it is possible that a machine with sufficient resources for a queued job will not process the job and stay idle instead. See Section 4.3.2 for further details on when a machine will choose to idle instead of processing a job.

4.3.1 Job Arrival

A two-level dispatching policy is used to assign arriving 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 $\varDelta_{jkl}$ 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 selected configuration. At the first level, no state information is required to make decisions. In the second level, the dispatcher makes 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 $\mathbf{A}^j_{ik} = 1$ 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. Of the machines in this configuration, a score of how far the current state of the machine is from the assigned bin is calculated for the class of the arriving job. Given the job class k, the machine j, the bin i that the machine emulates, and the current number of jobs of class k processing on the machine κ_{jk}, a score $\upsilon_{jk} = N_{ijk} - \kappa_{jk}$ is calculated. For example, if the bin has three jobs of class 1 ($N_{ijk} = 3$), but there is currently one job of class 1 being processed on the machine ($\kappa_{jk} = 1$), then $\upsilon_{jk} = 2$. The dispatcher will choose the machine with the highest υ_{jk} value that still has sufficient remaining resources to schedule the arriving job. In the case where no machines in the desired configuration are available, the dispatcher will use the roulette wheel selection method to choose another machine configuration with $\varDelta_{jkl} > 0$ that has not already been considered. If all configurations with $\varDelta_{jkl} > 0$ have insufficient capacity, the dispatcher will then check all remaining machines and immediately assign the job if one with sufficient idle resources is found. After all these checks, if there exists no machine that can immediately process the job, it will enter a queue belonging to the class of the job. Thus, there are a total of $|K|$ queues, one for each job class.

4.3.2 Job Exit

When a job completes service on a machine, resources are released and there is potential for new jobs to start service. The jobs that are considered for scheduling are those waiting in the job class queues. To decide which job to schedule on the machine, the dispatch policy will calculate the score υ_{jk} as discussed above, for every job class with $\varDelta_{jk} > 0$. We use the calculation of υ_{jk} to create a priority list of job classes where a higher score represents a class that we prefer to schedule first.

The scheduler considers the first class in the ordered list. The jobs in the queue are considered in the order of their arrival and if any job fits on the machine, the job is dispatched and υ_{jk} is decreased by one. While the change in score does not alter the ordering of the priority list sorted using υ_{jk}, the search within the queue will continue. If the top priority class gets demoted due to the scheduling of a job, then the next class queue is considered. This is continued until all classes with positive $\varDelta_{jk}$ values have been considered and all jobs in each of these queues cannot be scheduled onto the machine.

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 and its job mix.

4.3.3 Rationale for the Dispatching Policy

During a job arrival event, the roulette wheel selection method allows for the assignment to be probabilistically equivalent to the $\varDelta_{jkl}$ allocations while avoiding the necessity to obtain system state information. Note that using state information may improve selection by choosing a configuration that more accurately follows the prescribed $\varDelta_{jkl}$ values dynamically. However, there is a trade-off between gathering and maintaining the additional machine state information and the possible improvement due to reduced variability.

The second major decision for dispatching a job upon arrival is to assign it to a machine such that the mix of jobs on the machine fits the bin that the machine emulates. The method chosen is a simple count of the number of jobs that is compared to the bin's job mix, which we see as the most straightforward approach towards the goal of matching the bins. An alternative is to reason about the actual resources dedicated to the different job classes rather than the count of jobs. However, such an approach would require modeling the variance of resource requirements and developing a more complicated measure of bin emulation. As we currently see no obvious ways forward in this direction, we decided on our more straightforward approach.

Finally for the job arrival event, we check all other configurations before allowing a job to enter the queue because doing so allows for the exploitation of idle resources, even if they deviate from the guidance of the LP solutions. Such a deviation is beneficial because presence of idle resources means the system is likely to be in a lower capacity state, where responding to jobs immediately is more important than long-term efficiency. This policy attempts to schedule jobs immediately whenever possible to reduce response times, while biasing towards placing jobs in such a way as to mimic the bins which have been found to reduce the effects of resource fragmentation. Our policy does not preclude the assignment of a pairing between a job class and machine configuration with $\varDelta^*_{jkl} = 0$ when the system is heavily loaded, since the requirement is only that at least one machine has available resources. Specifically, if very few machines have sufficient idle resources, the scheduler may prefer queueing a job even though it can be immediately processed. However, the reasoning regarding when one should switch strategies is not clear and so the policy presented aims to simplify this decision by assuming that any time a machine has free resources, the scheduler will treat the system as though it were lightly loaded. A more nuanced approach may improve system performance, but we do not explore this detail.

The rationales for our choices in the job exit event are similar to the job arrival choices. By using a count of how the actual mix of jobs deviates from the emulated bin, the policy more closely mimics the chosen bin. Unlike the job arrival event, the choice of not scheduling any job classes with $\varDelta_{jkl} = 0$ is made since the system is likely heavily loaded (a queue has formed) and pairing efficiency has increased importance to improve system throughput. Therefore, it is possible that LoTES will idle a machine's resources even though a job in a queue with $\varDelta_{jk} = 0$ can fit on the machine because it is likely better to reserve those resources for a more efficient matching.

5 Experimental Results

We test our algorithm on real cluster workload trace data and on generated data. In this section, we provide details of our experiments. We start by presenting two scheduling algorithms we compare to our approach, followed by a discussion of the implementation challenges. We then describe and present results for the algorithms on the workload trace data and the generated data.

5.1 Algorithms for Comparison: A Greedy Dispatch Policy and the Tetris Scheduler

We consider two alternative schedulers: a Greedy policy and the Tetris scheduler. We chose to compare LoTES against the Greedy dispatch policy because it is a natural heuristic, which aims to quickly process jobs. Like the LoTES algorithm, the Greedy dispatch policy attempts to schedule jobs onto available machines immediately if a machine is found that can process a job. This is done in a first-fit manner where the machines are ordered following the list of machines in Table 1 (from top to bottom). In the case where no machines are available for immediate processing, the job enters a queue. Unlike the LoTES scheduler, since the Greedy dispatch policy does not make use of job class information, jobs enter a queue for a single machine. The policy will choose the machine with the shortest queue of waiting jobs with ties broken randomly. If a queue forms, jobs are processed in FCFS order.

The Tetris scheduler [7] aims to improve packing efficiency and reduce average completion time through use of a linear combination of two metrics. The packing efficiency metric is calculated by taking the dot product of the resource profiles of a job and the resource availabilities on machines. If we denote $\mathbf{r}$ as a vector representing the resource profile of a job and $\mathbf{C}$ as the resource profile for the remaining resources on a machine, then we can define the packing efficiency score as $\omega = \mathbf{r} \cdot \mathbf{C}$. A higher score represents a better fit of a job on a machine. The second metric, the amount of work, is calculated as the total resource requirements multiplied by the job duration. That is, given the processing time of a job p, the work score is $\gamma = p\mathbf{r} \cdot \mathbf{1}$, where $\mathbf{1}$ is a vector of ones. The Tetris scheduler prioritizes jobs with less work in order to reduce overall completion times of jobs. For our experiments, we give each of the metrics equal weighting and found that the relative performance of the Tetris scheduler does not improve with different weightings. The score for each job is then calculated as $\omega - \gamma$, where a larger score will have higher priority.

The Tetris scheduler addresses resource fragmentation through the use of the packing efficiency score. By placing jobs on machines with higher packing scores, machines with resource profiles that are similar to the job resource profiles are prioritized. Tetris benefits from being able to make packing decisions online, unlike LoTES. However, Tetris makes decisions myopically, without the foresight that new jobs will be arriving. In contrast, LoTES considers packing jobs in the long-term by generating bins in advance so that individual jobs may not share similar resource profiles as a machine, but the combination of jobs will be able to better make use of the resources of a machine.

Each scheduler requires different information to make scheduling decisions. All approaches make a scheduling decision when a job arrives to the system and use at least information regarding the resource requirements of a job along with the available resources of the machines. The Greedy scheduler maintains a queue for each machine and must also know the length of this queue. The Tetris scheduler requires the processing time of a job to calculate γ. Finally, LoTES uses job class data for the first two stages and the number of jobs from each class that are scheduled on a machine. In general, all these requirements can be obtained, if not exactly, then at least approximately. Although in practice, a scheduling model's performance is influenced by the accuracy of system data, we do not consider the sensitivity of the models to inaccuracies in data as each scheduler makes use of different information.

5.2 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, this delay may be an issue. For every new job arrival, the scheduler requires the currently available resources and the size of the queue of one or more machines. As the system becomes busier, the scheduler may have to obtain such information for all machines in the data center. Thus, scaling may be problematic as the algorithms may have to 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 as we see that the waiting time delays of jobs are orders of magnitude larger than the processing time. An additional issue may be present that could reduce performance as the scheduler itself creates additional load on the network connections within the data center. This may need to be accounted for if the network connections become sufficiently congested.

Note, however, that the dispatching overhead of arriving jobs for LoTES is no worse than that of the

Greedy policy or Tetris. The LoTES algorithm benefits from the restricted set of machines that it considers based on the Δ_{jk} values. At low loads where a job can be dispatched immediately as it arrives, the Greedy policy and LoTES will not have to gather state information for all machines. In contrast, the Tetris scheduler will always gather information on all machines to decide which has the best score. However, in the worst case, LoTES may require state information on every machine when the system is heavily loaded, just as the other algorithms.

A system manager for a very large data center must take into account the overhead required to obtain machine state information regardless of which algorithm is chosen. 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, 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, allowing the mappings from the offline stages of LoTES to still be used but with substantially less overhead for the online decisions.

5.3 Google Workload Trace Data

The first experiment we perform tests the algorithms on cluster workload trace data provided by Google.[3] These data represent the workload for one of Google's compute clusters over the one month period of May 2011, providing 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 the time for which 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 were to be processed on a different machine or if the load on the machine were to be different. Therefore, we assume that processing times are independent of machine configuration and load.[4]

Although the information provided is extensive, we limit what we use for our experiments to only the resources requested and duration for each job. We do not consider failures of machines or jobs: jobs that fail and are resubmitted are considered to be new, unrelated jobs. Figure 6 shows the number of jobs arriving during each hour for the entire month of the trace data. Machine configurations change over time due to failures, the acquisition of new servers, or the decommissioning of old ones, but we only use the initial set of machines and keep that constant over the whole month. Furthermore, system micro-architecture is provided for each machine and some jobs are limited in which types of architecture they can be paired with and how they interact with these architectures. 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.

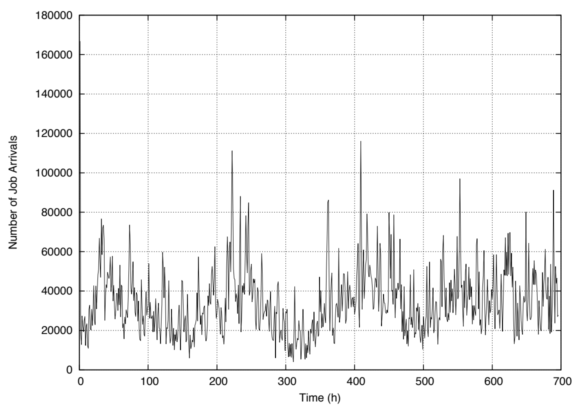

Fig. 6 The number of jobs arriving in each hour in the Google workload trace data.

5.3.1 Machine Configurations

The data center has 10 machine configurations as presented in Table 1. Each configuration is defined strictly by its resource capacities 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.

5.3.2 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. [20] by using k-means clustering to create job classes and use Lloyd's algorithm [16] to create

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

[4] We examine the impact of processing time variation in subsequent experiments (see Section 5.4.3).

# 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 for Google workload trace data.

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.

the different clusters. To limit the amount of information that LoTES is using in comparison to our benchmark algorithms, 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 and the Tetris scheduler do not use class information, any inaccuracies introduced by forming clusters in this way will only make LoTES worse when we compare the two algorithms.

The clustering procedure resulted in four classes being sufficient for representing most jobs. Increasing the number of classes led to less than 1% of jobs being allocated to the new classes. The different job classes are presented in Table 2. Although we only use the first day for determining the job class parameters, Figure 7 shows how the proportion of arriving jobs calculated is not constant for the entire data set. Rather, the values change heavily throughout the scheduling horizon.

5.3.3 Simulation Results

We created an event-based simulator in C++ to emulate a data center with the workload data as input. The LP models are solved using IBM ILOG CPLEX 12.6.2. We ran our tests on an Intel Pentium 4 CPU

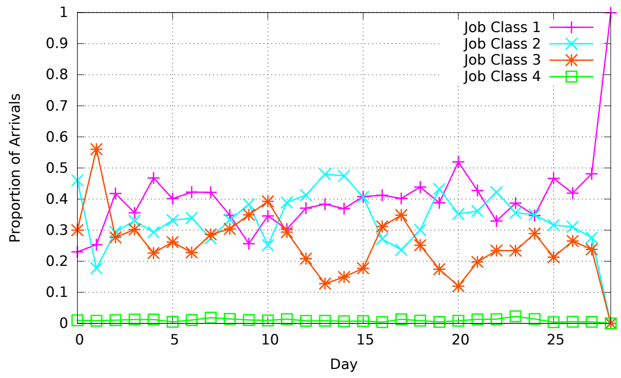

Fig. 7 Daily proportion of jobs belonging to each job class.

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 to obtain solutions to both LP models and generate bins is 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 workload. We leave this for future work.

Figure 8 presents the performance of the system over the one month period. The graph provides the mean response time of jobs on a log scale over every 24-hour interval. We include an individual job's response time in the mean response time calculation for the interval in which the job begins processing. We see that the LoTES algorithm greatly outperforms the Greedy policy and generally has lower response times than Tetris. On average, the Greedy policy has response times that are orders of magnitude longer (15-20 minutes) than the response times of the LoTES algorithm. The Tetris scheduler performs much better than the Greedy policy, but still has about an order of magnitude longer response times than LoTES.

The overall performance shows the benefits of LoTES, however, a more interesting result is the performance difference when there is a larger performance gap between the scheduling algorithms. In general, LoTES is as good as Tetris or better. However, when the two algorithms deviate in performance, LoTES can perform significantly better. For example, around the 200 hour time point in Figure 8, the average response time of jobs is minutes with the Greedy policy, seconds under Tetris, and micro-seconds with LoTES.

The Greedy policy performs worst as it is the most myopic scheduler. However, the one time period that it does exhibit better behaviour than any other scheduler is the first period when the system is in a highly tran-

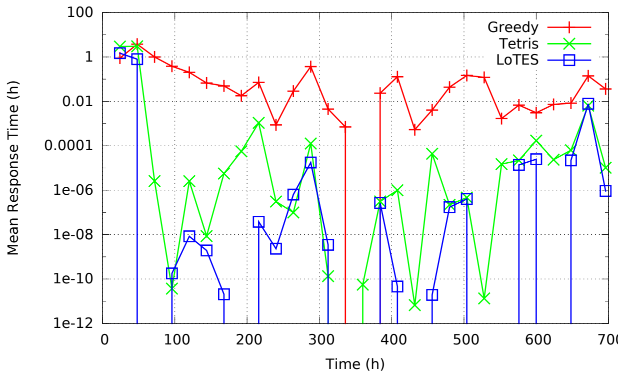

Fig. 8 Response Time Comparison.

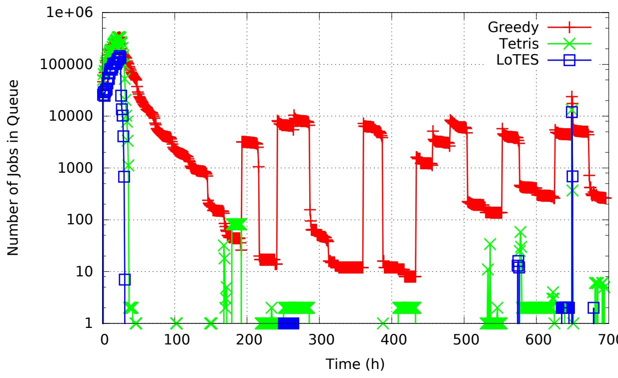

Fig. 10 Number of jobs in queue.

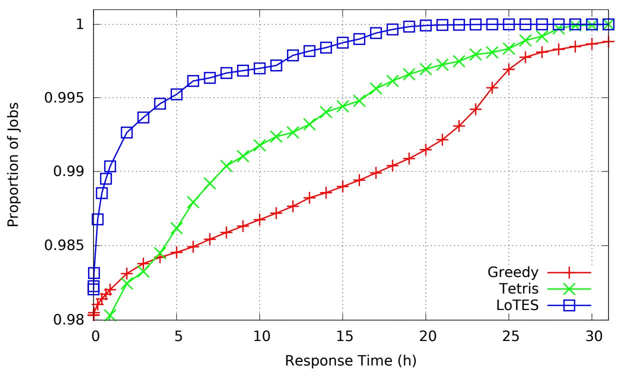

Fig. 9 Response time distributions.

sient state and is heavily loaded. We suspect this is also due to the scheduler being myopic and optimizing for the immediate time period which leads to better short-term results, but the performance then degrades over a longer time horizon.

Although it is shown in Figure 8 that LoTES can reduce response times of jobs, the large scale of the system obscures the significance of even these seemingly small time improvements between LoTES and Tetris. Often, the difference in average response times for these two schedulers is tenths of seconds (or even smaller). When examining the distribution of response times from Figure 9, we see that Tetris has a much larger tail where more jobs have a significantly slower response time. For the LoTES scheduler, less than 1% of jobs have a waiting time greater than one hour. In comparison, the Tetris scheduler has just as many jobs that have a waiting time greater than seven hours and the Greedy policy has 1% of jobs waiting longer than 17 hours. These values show how poor performance can become during peak times, even though on average, the response times are very short because the vast majority of jobs are immediately processed.

Finally, Figure 10 presents the number of jobs in queue over time. We see that for most of the month, the queue size does not grow to any significant degree for LoTES. Tetris does have a queue form at some points in the month, but even then, the queue length is relatively small. Other than at the beginning of the schedule, the throughput of jobs for Tetris and LoTES is generally maintained at a rate such that arriving jobs are processed immediately. The large burst of jobs early on in the schedule is due to the way in which the trace data was captured: all these jobs enter the system at the beginning as a large batch to be scheduled. However, as time goes on, these initial jobs are processed and the system enters into more regular operation. The Greedy policy on the other hand has increased queue lengths at all points during the month.

Given that, for the majority of the scheduling horizon, LoTES is able to maintain empty queues and schedule jobs immediately, we found that a scheduling decision can often be made by considering only a subset of machine configurations rather than all machines in the system. In contrast, the Tetris scheduler, regardless of how uncongested the system is, will always consider all machines to find the best score. We do not present the scheduling overhead, but it is apparent from the graph that without a queue build up, the overhead of LoTES will be no worse, and more likely better, than Tetris.

It is important to state here again that LoTES makes use of additional job class information, which is not considered by the other schedulers. However, the information can be inaccurate as seen in Figure 7, where the proportion of arriving jobs belonging to a job class can be seen to change over time. One would expect that improvements could be made by dynamically updating the parameters of the job classes to ensure that LoTES maintains an accurate representation of the system. Regardless, even with a fairly naive approach where the job classes defined are assumed to be static, the LoTES scheduler is able to perform well.

5.4 Randomly Generated Workload Trace Data

Randomly generated data is used to show the behaviour of LoTES when we vary the resource requirements of job classes and include machine dependent processing times.

In two experiments, we have nine job classes that all arrive at the same rate $\alpha\lambda$, where $\alpha = \frac{1}{9}$ and λ is the total arrival rate of the system. Jobs arrive following a Poisson process with exponentially distributed inter-arrival times. Each job, z, has an amount of work, w_z that must be done, which is generated from an exponential distribution with mean one. The work will be used to define the processing time as $p_z = \frac{w_z}{\mu_{jk}}$ given that job z is a job of class k and is processed on a machine of configuration j. To generate the resource requirements of a job, a randomly generated value following a truncated Gaussian distribution with mean r_{kl}, coefficient of variation 0.5 for class k and resource l, and truncated to be in the interval [0, 1], is obtained for each resource $l \in R$.

5.4.1 Machine Configurations

We use the same machine configurations from the Google workload trace data in Table 1, except we change the total number of machines in each configuration. We use 1000 machines per configuration so that the system is more equally balanced between the different types of configurations available. Although balancing the configurations is not crucial, it is done to emphasize the heterogeneity of machines; more specifically, we wish to avoid having one or two configurations that represent the majority of all machines in the system.

5.4.2 Job Class Details: Varying Resource Requirements

The first set of generated data we test varies the resource requirements between different classes. We consider a range of systems starting with one where all nine job classes have the same resource requirement distribution and progressively increasing the differences between the job classes.

We define the parameter Φ to denote the measure of the difference in resource requirements of job classes. Given some value Φ, we randomly generate a value for each job class and resource pair $\phi_{kl} = U[-\Phi, \Phi]$ following a uniform distribution. Jobs from class k will then have resource requirements generated from a truncated Gaussian distribution with mean $r_{kl} = 0.025 + \phi_{kl}$, coefficient of variation of 0.5, and truncated to be in [0,1]. As Φ grows, we expect to see larger differences between the resource requirements of jobs between different classes. When $\Phi = 0$, all job classes have the same resource requirement distribution.

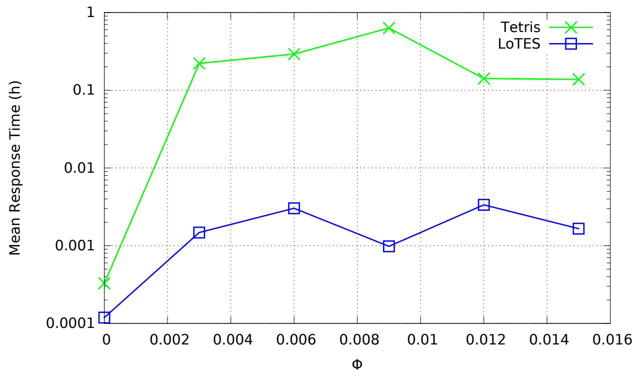

Fig. 11 Results for varying resource requirements between job classes.

We choose an arrival rate of $\lambda = 0.97\lambda^*$, where λ^* is the solution of the machine assignment LP. This load represents a heavily utilized system that is, from preliminary experiments, still stable for LoTES and Tetris.[5] However, we found that the Greedy policy is not stable: queue sizes increase unboundedly with time. Therefore, we only show results for LoTES and Tetris.

Simulations are done for values of Φ between 0 and 0.015, in increments of 0.003. Thus, the systems we test range from one where all mean resource requirements are 0.025 regardless of job class or resource, to one that can have average resource requirements ranging from 0.01 to 0.04. The processing rate is generated by first obtaining a uniformly distributed random value $u_k = U[0, 1]$ for each job class k, and setting $\mu_{jk} = u_k^{-1}$ for all machine configurations j. For each value of Φ, we generate five different instances, by generating r_{kl} and u_k values independently, and simulating the system for 100 hours. The mean response time for all jobs in the 100 hour simulation is recorded and the mean over the five instances for each tested Φ is presented in Figure 11.

When $\Phi = 0$, all job classes are the same and we see that both scheduling algorithms yield short response times. Due to the logarithmic scaling of our graph, the apparent difference is actually insignificant.

As Φ increases, we see that both scheduling algorithms have longer response times. We believe this to be due to the fact that the maximum system load, λ^* becomes looser as Φ grows due to fragmentation and wasted resources. This issue is further exacerbated by

[5] Note that λ^* represents an upper bound on the system load that can be handled. The bound may not be tight depending on the fragmentation of resources on a machine and/or the inefficiencies in the scheduling model used.

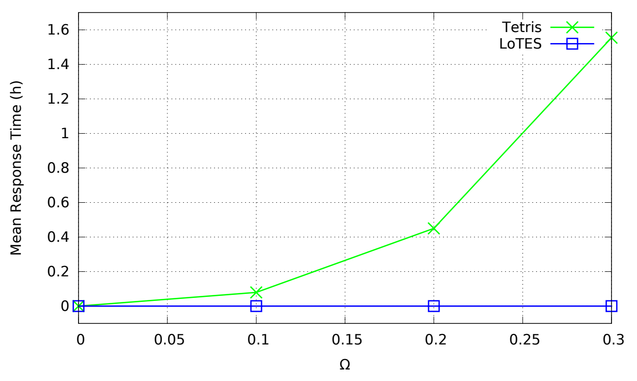

Fig. 12 Results for varying processing time between job classes. System load of 0.90.

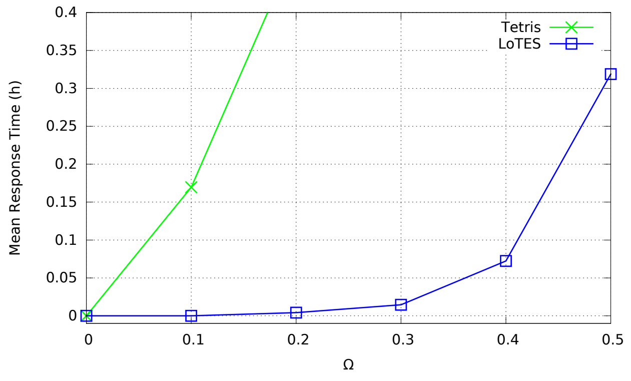

Fig. 13 Results for varying processing time between job classes. System load of 0.95.

the inefficiencies in scheduling that decrease the throughput of machines, effectively increasing the system load. Thus, we see that both scheduling models have longer response times when $\Phi > 0$, and that Tetris becomes much worse than LoTES. LoTES takes better advantage of efficient packing of jobs onto machines using the allocation LP and machine assignment LP solutions.

5.4.3 Job Class Details: Varying Processing Time

The second set of generated data we consider looks at processing times that are dependent on the machine configuration. For these experiments, we use the resource requirements, r_{kl}, generated from the previous experiment with $\Phi = 0.06$. Rather than using a random value u_k to obtain the processing rate, we include an additional value ω_{jk}, a multiplier that makes the processing rate dependent on the machine configuration. Given some value Ω, we randomly generate ω_{jk} from a uniform distribution $U[1-\Omega, 1+\Omega]$ for each machine configuration j and job class k. The processing rate is then calculated as $\mu_{jk} = u_k\omega_{jk}$.

We test a range of Ω values to observe how the scheduling models behave as we change from a system with machine-configuration-independent processing times to ones with increased machine configuration dependency. As before, five instances are generated for each value of Ω where we use the same r_{kl} values from the previous experiment, but generate ω_{jk} independently for each instance. A simulation time of 100 hours is performed and the mean response time is recorded.

We consider two different system loads: 0.90 and 0.95. Both these loads are chosen to be lower than in the previous experiment as we found from preliminary experiments that a load of 0.97 often led to instability in the system. Figures 12 and 13 show the system performance with loads of 0.90 and 0.95, respectively. We do not present results for Greedy as we found at these loads, the system is not stable. At a load of 0.95, we also found that Tetris appears to be unstable at higher values of Ω and thus response times are only reported for $\Omega \leq 0.1$.

At a load of 0.90, LoTES is essentially able to start all jobs immediately upon arrival. In comparison, Tetris is able to start all jobs immediately when $\Omega = 0$, but we see a continual increase in the average response time as Ω increases, as scheduling inefficiencies result in a drastic reduction of system throughput. To illustrate the performance of LoTES with increased Ω, we test a system load of 0.95 so that LoTES is no longer able to immediately start all jobs. Similar to Tetris, we see a rapid growth in response time with Ω. We suspect that the reason that LoTES outperforms Tetris on these experiments is due to its ability to find efficient allocations that take into account the trade-off between processing time dependencies and fragmentation due to job mixes. Tetris also considers processing time dependencies and job fragmentation, but does so greedily by prioritizing low processing time allocations and best-fits of the resource requirements rather than efficient mixes. Incorporating longer term reasoning that considers the system performance rather than the job performance means that the LoTES algorithm is better equipped to handle varied processing times as it can make informed decisions on a set of jobs.

6 Conclusion and Future Work

In this work, we developed the LoTES scheduling algorithm that improves response times for large-scale data centers by creating a mapping between jobs and machines based on their resource profiles. The algorithm consists of three stages:

1. A queueing model uses a fluid representation of the system to allocate job classes to machine configurations. This stage extends existing models in the queueing theory literature to include multi-capacity resources and provides long-term stochastic knowledge by finding efficient pairings of job classes and machine configurations that lead to maximizing system throughput for the abstracted system.
2. A stage that assigns a particular job mix to each machine. The assignment is restricted by the solution of the first stage in order to both reduce the combinations that are considered and to incorporate the long-term view of the system. This stage treats jobs and machines as discrete entities and performs combinatorial reasoning without losing the long-term knowledge.
3. A dispatching policy to realize the machine assignments made in the second stage. The primary goal of this stage is to ensure that the system tends towards scheduling decisions that will have machines processing a set of jobs similar to the job mixes assigned in Stage 2. However, the policy also aims to reduce response times by actively deviating from the prescribed assignments when the system has idle resources. This stage allows for the scheduling system to respond to the incoming arrival of tasks in a timely manner while benefiting from the offline optimization.

Our algorithm was tested on Google workload trace data and on randomly generated data, where we found it was able to reduce response times by orders of magnitude when compared to a benchmark greedy dispatch policy and by an order of magnitude when compared to the Tetris scheduler [7]. We believe that the main advantage of LoTES over Tetris is that the former considers future job arrivals by generating efficient bins in advance, which can then be mimicked by the machines online. LoTES behaves less myopically and can reason about good packing efficiency based on combinations of jobs rather than a single job at a time. This improvement is also computationally cheaper during the online scheduling phase since LoTES often requires state information for fewer machines when making assignment decisions.

The data center scheduling problem is very rich from the scheduling perspective and our approach 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, the offline phase is sufficiently fast (about 1 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, allowing LoTES to learn to predict the expected mix of jobs that will arrive to the system and make scheduling decisions with these predictions in mind. Not only do we wish to be able to adapt to a changing environment, but we also wish to extend our algorithm to be able to more intelligently handle situations when the mix of jobs varies greatly from the expectation. Large deviations from the expectation will lead to system realizations that differ significantly from the bins created in the second stage of the LoTES algorithm and make the offline decisions less relevant to the realized system.

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, users may under or overestimate their resource requirements and the utilization of a resource may change over the duration of the job itself. Uncertainties in resource usage add difficulty to the problem because instead of knowing the exact amount of requested resources once a job arrives, we only have an estimate and must ensure that a machine is not underutilized or oversubscribed.

Finally, the literature on data center scheduling has considered a various different objectives and constraints. Fairness among multiple users has been an important topic to ensure that the system not just responds quickly to job requests, but provides equal access to resources [11,29]. We would like to include fairness considerations using LoTES, which can be accomplished by either including users in the LP models of the first two stages to ensure resources are shared, or by introducing prioritization for fairness in the dispatch policy of the third stage in a similar way as Delay scheduling [29]. Another important system aspect is energy consumption [3,15]. Tarplee et al. [26] present a multi-stage scheduling model similar to LoTES that directly considers energy consumption in a data center, where jobs do not arrive dynamically over time (as they do in our system). Their scheduler uses an LP relaxation with similar goals to ours in that it relaxes the problem to allow the ability to divide the load of a job across multiple machines. The LP solution then is used to guide the scheduling choices. The minimization of energy consumption is crucial for running low-cost data centers and is an important area for future work.

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). We also wish to thank the referees for their insightful comments and providing directions for additional work which has resulted in this paper.

References

1. 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)
2. Andradóttir, S., Ayhan, H., Down, D.G.: Dynamic server allocation for queueing networks with flexible servers. Operations Research **51**(6), 952–968 (2003)
3. 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)
4. 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)
5. 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)
6. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: Proceedings of the 8th USENIX conference on Networked systems design and implementation, vol. 11, pp. 323–336 (2011)
7. 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)
8. Guazzone, M., Anglano, C., Canonico, M.: Exploiting vm migration for the automated power and performance management of green cloud computing systems. In: Energy Efficient Data Centers, vol. 7396, pp. 81–92. Springer (2012)
9. Guenter, B., Jain, N., Williams, C.: Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In: INFOCOM, 2011 Proceedings IEEE, pp. 1332–1340. IEEE (2011)
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. Jain, R., Chiu, D.M., Hawe, W.: A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Digital Equipment Corporation Research Technical Report TR-301 pp. 1–37 (1984)
13. 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)
14. 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)
15. 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)
16. Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information Theory **28**(2), 129–137 (1982)
17. 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)
18. 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)
19. Mann, Z.Á.: Allocation of virtual machines in cloud data centers–a survey of problem models and optimization algorithms. ACM Computing Surveys **48**(1), 1–31 (2015)
20. 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)
21. 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)
22. Rasooli, A., Down, D.G.: COSHH: A classification and optimization based scheduler for heterogeneous hadoop systems. Future Generation Computer Systems **36**, 1–15 (2014)
23. 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)
24. Salehi, M.A., Krishna, P.R., Deepak, K.S., Buyya, R.: Preemption-aware energy management in virtualized data centers. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pp. 844–851. IEEE (2012)
25. 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)
26. Tarplee, K.M., Friese, R., Maciejewski, A.A., Siegel, H.J., Chong, E.K.: Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques. IEEE Transactions on Parallel and Distributed Systems **27**(6), 1633–1646 (2016)
27. Terekhov, D., Tran, T.T., Down, D.G., Beck, J.C.: Integrating queueing theory and scheduling for dynamic scheduling problems. Journal of Artificial Intelligence Research **50**, 535–572 (2014)
28. 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)

29. 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)