Report: Google Data center Scheduling

Tony T. Tran and **J. Christopher Beck**
Department of Mechanical and Industrial Engineering, University of Toronto
{tran, jcb}@mie.utoronto.ca

1 Research Objective

The purpose of our research project is to provide scheduling algorithms for large-scale data centers. The size of the data centers typically deployed by companies like Google are immense and so classical scheduling techniques are limited in use. Further, the highly dynamic nature of the environment creates additional complications in a pursuit of optimizing the system. Previous work has shown the advantages of combining queueing theory and scheduling to provide richer modelling of a complex system which contains both dynamic and combinatorial aspects. We wish to continue this line of research and apply stochastic modelling techniques with combinatorial optimization in a goal to develop a scheduling algorithm for a large-scale data center.

2 Introduction & Motivation

The advancement of science and engineering has led to more complicated and demanding computational requirements. Many computational jobs are of such massive size that a personal workstation is insufficient. Not only are the needs of the scientific community growing, so are those of the general population. The cloud computing paradigm of providing hardware and software remotely to end users has become very popular with such applications as e-mail, Google documents, iCloud, and dropbox. The organizations that supply these resources must employ large data centers to provide end users with the services requested. As the demand for computational resources increases, so must the supply of services efficiently scale.

The problem proposed by Google is the efficient scheduling of jobs in a large-scale data center. A typical data center at Google can contain tens of thousands of machines. The efficient and fast scheduling of jobs in the system to the massive number of machines must be automated to gain the benefits of using such a large data center. By efficiently scheduling jobs, one could expect to see increased performance using the resources already available. This leads to either better service provided or the potential for delaying the purchase of new hardware to meet increased demand.

Building on recent work, we investigate the potential of combining queueing theory and scheduling to provide a unified model which accommodates the high level and low level aspects of the problem. In particular, Terekhov et al. [1] showed the ability to apply queueing analysis to scheduling algorithms to prove the conditions of which the algorithm is well behaved. Further, the paper looks at the performance measure of mean flow time and show that considering both long run and short run aspects of the problem can lead to better performance. An extension of the paper by Terekhov et al. [2] examine the combination of queueing theory and scheduling on an algorithmic level and add a hybrid model which takes aspects of queueing theory and scheduling. The hybrid model is found to perform very well in comparison to the pure queueing theory or scheduling models. As such, we suspect that using ideas from both queueing theory and scheduling can lead to good performance when scheduling a data center, which contains a high level of dynamic and combinatorial properties.

3 Problem Definition

The problem deals with assigning jobs to machines. There are two types of jobs defined by Google: batch jobs and service jobs. Service jobs consume resources over a long period of time and have priority over batch jobs. The resources that remain after considering all service jobs are allocated to batch jobs. In comparison, batch jobs

are much shorter and will only consume resources from a few minutes to hours. In our study, we focus on batch jobs and ignore service jobs. Thus, we restrict ourselves to the problem of scheduling these faster jobs that arrive more stochastically.

Jobs contain tasks, which are sent to machines to be processed. A task consumes a given amount of resources on the machine for the duration of its processing time. Once the machine has finished processing a task, it leaves the machine and the resources are free to be used by other tasks. The following sections describe the characteristics of jobs and machines, and also our assumptions about the system in greater detail.

Jobs

Jobs arrive at a rate of λ (number of jobs per second). Each job belongs to one of a set of K classes with probability α_k. A job can contain multiple tasks, which are the smallest units of items that are processed on the server cluster. We assume that the arrivals are independent and identically distributed (i.i.d.) as well as time independent. A job's class defines the distribution of resource requirements of each type. We denote the expected amount of resources of type l required by class k as r_{kl}. A job consumes the amount of required resources for the duration of its processing time. The processing time for class k is assumed to be i.i.d. with service rate μ_k.

A job can be allocated resources from several different machines. However, resources allocated to a task cannot be split over multiple machines. All tasks require a positive amount of cores and memory. We currently assume that jobs within a job class have the same resource requirements distribution (normally distributed with coefficient of variation CV, where $CV \in (0, 1]$).

Machines

In practice, commonly used resources are processing cores, random access memory (RAM), disk-space, and bandwidth. For our study, we generalize the system to have $|R|$ resources, where R denotes the set of all resources. There are many machines that work in parallel to process jobs in the data center. We are interested in dealing with a server cluster that has on the order of 10,000 machines. The entire machine population is divided into approximately 10 different configurations. Each machine configuration belongs to the set of M configuration types which are defined by the capacity of every resource available to that machine. A machine of configuration j will have c_{jl} amount of resource l and within a configuration j there are n_j identical machines.

3.1 Objectives

There are several objectives that were initially presented as relevant issues for data center scheduling. We were told to consider the follow:

1. Minimal wasted resources to guarantee the operation of service jobs.
2. Minimal wasted work due to preemptions that require jobs to be restarted.
3. Minimal queueing time of batch jobs.
4. Maximal goodput of batch jobs. Goodput is the effective throughput of batch jobs, which is (useful output / time).

Objectives 1 and 2 are not relevant to our current study. These goals arise because of the existence of two types of jobs: batch jobs and service jobs. Since we will only consider the scheduling of batch jobs, the first two objectives will not be our focus.

Objective 3 is just the standard scheduling performance measure of response time or waiting time. For performance conscious users, the timely execution of a requested job is a very important quality of service expected.

Even users without strict time constraints will feel burdened by any large delays between the time of a job submission and actual execution. In today's technologically advanced society, online search queries are distinguished by the millisecond and scrutinized for even indistinguishable delays in comparison to competitors.

The final objective of goodput can be likened to the classical queueing theory performance metric of throughput. Not only does the goodput have correlations to response time performance, but efficiently increasing goodput can alleviate demands on the data center to increase capacity. Given the same data center configurations, if a scheduling algorithm is able to increase the effective goodput of batch jobs, then an inflation in demand can be met with a more delayed response to purchasing requirements for additional resources.

4 LoTES Model

The proposed model, Long Term Evaluation Scheduling (LoTES, pronounced "lotus"), is a three-stage queueing theory and optimization hybrid approach. 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 tasks to machines and is performed online. In the first stage, we update the allocation LP presented by Andradóttir et al. [3] to account for multiple resources and use it to represent the data center as a fluid model. Such a representation finds 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 values found from solving the allocation LP. In the final stage, jobs will be dispatched to machines dynamically as they arrive to the system.

4.1 Allocation LP

Andradóttir et al.'s [3] allocation LP is created for a similar problem except with a single unary resource. The allocation LP finds the maximum arrival rate of jobs that a given queueing network can handle while maintaining stability. A stable system is defined as a system where the expected number of jobs present is finite as time goes to infinity.

In our problem, there are $|R|$ resources which must be accounted for on each machine and with each job. We update the allocation LP to accommodate multiple resources. The model considers each machine configuration's resources in an aggregated manner to pool all available resources from the same machine configuration into a single super machine. The model then ignores the issue of resource fragmentation within machines of a single configuration. Fragmentation occurs when the machine's resource capacity cannot be fully utilized when machines are treated individually. For example, if a machine configuration has 30 machines with 8 cores available on each machine and a set of jobs assigned to the configuration requires exactly 3 cores for each task, the total pool of resources would be 240 processors that can process 80 tasks in parallel. If the machines were considered as discrete and independent machines, only 2 tasks could be placed on a single machine since a third task on a machine would require it to have 9 cores. Therefore, the remaining 2 cores are wasted and in reality only 60 tasks can be processed in parallel. The effect is further amplified when multiple resources exist. Instead of the possibility of wasting just cores, we may also waste resources of other types because of the core bottleneck. Since a fluid representation obtains an upper bound on the possible arrival rates that the system can handle, the subsequent stages of the LoTES model heuristically deal with fragmentation problems.

The updated allocation LP is shown below.

$$\max \quad \lambda$$

$$\text{s.t.} \quad \sum_{j \in M} (\delta_{jkl} c_{jl} n_j) \mu_k \geq \lambda \alpha_k r_{kl} \qquad k \in K, l \in R \qquad (1)$$

$$\frac{\delta_{jkl} c_{jl}}{r_{kl}} = \frac{\delta_{jk1} c_{j1}}{r_{k1}} \qquad j \in M, k \in K, l \in R \qquad (2)$$

$$\sum_{k \in K} \delta_{jkl} \leq 1 \qquad j \in M, l \in R \qquad (3)$$

$$\delta_{jkl} \geq 0 \qquad j \in M, k \in K, l \in R \qquad (4)$$

where,

Decision Variables

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

Parameters

- α_k: Probability of a job being in class k
- μ_k: Service rate of tasks that belong to batch jobs in class k
- c_{jl}: Capacity of resource l on machine type j
- r_{kl}: Expected requirement of resource l for jobs of class k
- n_j: Total number of machines of configuration type j

The LP presented here assigns the fractional amount of resources that each machine configuration type should allot to classes with the goal of maximizing the capacity of the system. Constraint (1) guarantees that enough resources are allocated for the requirements of each class. Constraint (2) ensures that the resources that are assigned to a job mimic the resource requirement profiles. The allocation LP prevents assigning more resources than available with constraint (3). Finally, constraint (4) 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, without accounting for fragmentation, the allocation LP only acts as an upper bound on the achievable capacity of a system rather than a tight bound which Andradóttir et al. [3] are able to achieve for the unary resource problem. Therefore, with the simple 30 machine system shown above, the allocation LP would assume that 80 tasks is the maximum that can be handled, but the system can only truly handle 75% of that load. In a real system, we would expect to see much better performance from the allocation LP. Resource requirements will vary, and this variance would allow for a scheduler to place smaller jobs into the wasted resources of the example where all task's core requirements were fixed to 3. Regardless, the solution found from the allocation LP provide insight into what a scheduler could consider to be good assignment decisions.

4.2 Machine Assignment

The allocation LP assigns a portion of each machine configuration's resources to the different job classes. The assignments are made using a fluid representation of the system where the machines of the same configuration are pooled. In the second stage, we perform a machine assignment that determines how each individual machine should serve different job classes in order to best mimic the solution found by the allocation LP.

The goal when assigning individual machines to specific job classes is to make assignments in such a way that the δ^*_{jkl} values are closely mimicked with the goal of maximizing the arrival rate that the system can stabilize. In

the second stage, we are concerned with fragmentation and attempt to assign machines to discrete jobs. Since the stage is performed offline, the jobs we assign can not be realized jobs that have arrived to the system. Rather, we will use the expected resource requirement of each job class to make assignments.

The allocation LP provides us with the classes that will be assigned to each machine configuration. If any of the δ^*_{jkl} variables are non-zero, then we know that at least some of the resources from a configuration j must be allocated to class k. The machine assignment algorithm will first consider the machine resource capacity and job class resource requirements to create multiple bins. A bin represents a snapshot in time of a machine with a set of tasks that can be processed on the machine in parallel. Once a complete set of bins are created to represent all assignments of tasks 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 generated bin.

The methodology to generate all possible non-dominated bins is shown in Algorithm 1. Given K^j, a set for machine configuration j containing all classes with positive δ^*_{jkl}, and a set b^j containing all possible bins, Algorithm 1 is performed for each machine configuration j. We make use of two functions not defined in the pseudocode:

- sufficientResource(K^j_i, b^j_y): Returns true if bin b^j_y has sufficient remaining resources for a task K^j_i.
- mostRecentAdd(b^j_y): Returns the job class of the task that was most recently added from those currently in b_y.

Algorithm 1 Generation of all non-dominated bins

```
y ← 1
x ← 1
x* ← x
nextBin ← false
while x ≤ |K^j| do
  for i = x* → |K^j| do
    while sufficientResource(K^j_i, b^j_y) do
      b^j_y ← b^j_y + K^j_i
      nextBin ← true
  x* ← mostRecentAdd(b^j_y)
  if nextBin then
    b^j_{y+1} ← b^j_y − K^j_{x*}
    y ← y + 1
  else
    b^j_y ← b^j_y − K^j_{x*}
  if b^j_y == NULL then
    x ← x + 1
    x* ← x
  else
    x* ← x* + 1
```

The constraint on the tasks in a bin is that the total resources required by the set of tasks must not exceed the total resources available on a single machine. It is possible for one bin to dominate another, meaning that the set of tasks in the latter bin is a subset of the tasks contained in the first bin. Ideally we want to have as few cases of domination as possible in our algorithm to eliminate redundancy and solve time. Dominated bins are considered inefficient since the dominating bin processes the same set of tasks while also processing one or more additional tasks.

Algorithm 1 starts by greedily filling the bin with tasks from a class. Once full, it will then move on to the next class of jobs and attempt to continue filling the bin whenever possible. Once no more tasks from any class are able to fit, we know that the bin is a dominating bin. The algorithm then backtracks by removing the last task added and tries to add a task from another class. This continues until the algorithm has exhaustively searched for all non-dominated bins.

Since the algorithm is 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 tasks we can fit of class k onto the machine of interest, than in the worst case, we must consider $\prod_{k \in K} L_k$ bins. We can improve the performance of the algorithm by ordering the classes in decreasing order of resource requirement. Of course, this measure is difficult to adhere to because there exists multiple resources. One would have to ascertain the constraining resource on a machine and this may be dependent on which mix of tasks are used.

Although the upper bound on the number of bins to consider is very large, we are able to find all dominating bins quickly given our problem size. The main reason we are able to find these bins within one second is because the the allocation LP helps to reduce the number of classes in consideration. Once we've solved for the δ^*_{jk} values, we find that we generally only see between one and four job classes ever assigned to a machine. When solving Algorithm 1, the number of bins generated are in the thousands of bins. Without the allocation LP, we find that there are on the order of millions of bins. However, even solving these problems can be done within a few minutes.

With the created bins, machines are then assigned to emulate one of the bins in order to best represent the allocation LP solution. To mimic the δ^*_{jkl} values, we must find the contribution that each bin makes. A bin i from machine configuration j will contain a certain number of tasks, N_{ijk} from a class k. Using the expected resource requirements, we can calculate the amount of resources l on machine j that contributes toward servicing jobs of class k, denoted ϵ_{ijkl} as $N_{ijk} r_{kl}$. The machine assignment LP is then

$$\begin{aligned}
\max \quad & \lambda \\
\text{s.t.} \quad & \sum_{j \in M} \Delta_{jkl} \mu_k \geq \lambda \alpha_k r_{kl} && k \in K, l \in R && (5) \\
& \sum_{i \in B_j} \epsilon_{jkl} x_{ij} = \Delta_{jkl} && j \in M, k \in K, l \in R && (6) \\
& \sum_{i \in B_j} x_{ij} = n_j && j \in M && (7) \\
& x_{ij} \geq 0 && j \in M, i \in B_j && (8)
\end{aligned}$$

where,

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 the configuration type j

Parameters

ϵ_{ijkl}: Fractional contribution of resource l assigning a machine to bin type i will have on class k

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 the system can handle. Constraint (5) is the equivalent to Constraint (1) of the allocation LP while accounting for discrete machines. The constraint ensures that a sufficient amount of resources are available to maintain stability for each class of jobs. Constraint (6) assigns the amount of total resources of l from machine j 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 (7). Finally, constraint (8) forces x_{ij} to be positive.

4.3 Dispatching Tasks

In the third and final stage of scheduling the batch jobs, a two-level dispatching algorithm is used to assign tasks to machines as they arrive to the system. The goal of the dispatching algorithm is to use the machine assignments in an attempt to emulate the proposed bins of the second stage. In the first level of the dispatcher, a task will be 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 are assigned to each machine configuration. In the second level of the dispatcher, the task is to be placed on one of the machines in the configuration it was assigned.

Deciding which machine configuration to assign a task can be done by revisiting the total amount of resources each configuration contributes to a job class. We can compare the Δ_{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 $\sum_{j=1}^{|M|} \Delta_{jkl}$ resources of type l, a machine configuration j should serve $\rho_{jk} = \frac{\Delta_{jkl}}{\sum_{m=1}^{|M|} \Delta_{mkl}}$ portion of the total jobs in class k. Therefore, to decide which configuration to assign an arriving task of job class k, 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 task 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 a $n_j \times |K|$ 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 tasks of class k assigned. $\mathbf{A}^j$ indexes which machines in the data center can serve a job of class k which we use to make our scheduling decision.

At the first level, no state information was used to make decisions. However, in the second level, we will make use of the exact resource requirements of a job as well as the state of machines in the system to make a decision. The dispatcher will attempt to dispatch the job to a machine of the configuration assigned from the first step. The dispatcher will make use of a first fit policy where the first machine found from those with $\mathbf{A}_{ik}^j = 1$ that has the available resources for the task will begin immediate service. In the case where no machines are available, the task will search through all other machines in the data center for immediate processing. If a machine exists with enough resources to immediately process the task, it will begin servicing the task. Otherwise, the task will enter the smallest queue of the machines belonging to the configuration assigned in the first step with $\mathbf{A}_{ik}^j = 1$. Where there are ties, the job will be dispatched randomly to one of the tied queues. By allowing for the dispatcher to make assignments of tasks to machines with $\delta_{jkl}^* = 0$, we enable free resources to be used immediately. One could expect that a system not at heavy loads 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. Following such a dispatch policy attempts to schedule jobs immediately whenever possible with a bias towards placing tasks on bins which have been found to be efficient.

5 Experimental Results

In this section, we present the results of simulating the LoTES model along with the experimental details. We use a discrete event simulation to test the response time performance of the proposed model against a naive scheduler that greedily assigns tasks to the machine with the shortest queue with sufficient resources to handle the job. Once assigned, the jobs will take on a first in, first out (FIFO) sequence. This greedy scheduler acts as a resource-unaware algorithm to compare against LoTES which can reason about the heterogenous machines.

Configuration	# of Machines	Core Capacity	RAM Capacity
1	1000	4	2
2	1000	4	4
3	1000	4	8
4	1000	4	16
5	1000	8	2
6	1000	8	4
7	1000	8	8
8	1000	8	16
9	1000	8	32
10	1000	24	32

Table 1: Data center machine configuration

5.1 Experimentation

A typical Google data center may contain 10,000 servers which are not all identical. We can expect a data center to be made up of about 10 different configurations of machines, where all machines in a configuration will share a tuple of characteristics such as: number of cores, amount of random access memory (RAM), storage space, network bandwidth, system architecture, etc. We will test our algorithm using a data center with the setup defined in Table 1.

The jobs that arrive to the system will require various amounts of each resources from a server. Mishra et al. [4] examined trace data from Google's compute cluster and found that the limiting resources were cores and RAM. Therefore, we will focus on these two resource requirements in this report and assume no other resources are ever a constricting factor in scheduling. Further, Mishra et al. [4] partition jobs into 8 different classes which define the expected core and RAM requirements as well as the processing times of jobs. The classes in Mishra et al.'s paper are defined by categorizing each of the three parameters based on relative sizes: small, medium, and large. However, their investigation leads to the discovery that processing times of the jobs in a Google compute cluster are largely bimodal and are either small or very large. They also show that all 27 possible combinations are not required and that the coefficient of variation is less than one when using only 8 classes. In our simulation, we will make use of the classes presented in their work which is presented in Table 2. The definition of each classification can be found in Table 3 which is a rough approximation of Mishra et al.'s [4] data.

Class	Processing Time	Cores	RAM
1: sss	small	small	small
2: sm*	small	medium	all
3: slm	small	large	small + med
4: sll	small	large	large
5: lss	large	small	small
6: lsl	large	small	large
7: llm	large	med+large	small + med
8: lll	large	med + large	large

Table 2: Job classification index

Using the data from Table 3, we will generate arriving jobs to the system and simulate their traversal through the system using our algorithm. Jobs arrive as a Poisson process and processing times of the tasks are exponentially distributed with the expected values found from Mishra et al.'s work. Resource requirements are generated as a

		Processing Time (hrs)		Processing Cores			RAM		
Class	Proportion	Expected	Range	Expected	Range	CV	Expected	Range	CV
1	0.33	0.083	0 - 2	0.08	0 - 0.2	0.16	0.48	0 - 0.5	0.09
2	0.29	0.32	0 - 2	0.4	0.2 - 0.5	0.03	0.74	0 - 32	0.09
3	0.06	0.65	0 - 2	1.11	0.5 - 4	0.03	0.68	0 - 1	0.08
4	0.01	0.42	0 - 2	1.39	0.5 - 4	0.04	1.54	1 - 32	0.05
5	0.17	18.34	2 -24	0.12	0 - 0.2	0.19	0.48	0 - 0.5	0.11
6	0.02	22.23	2 - 24	0.16	0 - 0.2	0.09	1.66	1 - 32	0.09
7	0.07	20.49	2 - 24	1.22	0.2 - 4	0.03	0.65	0 - 1	0.08
8	0.05	18.85	2 - 24	1.32	0.2 - 4	0.06	1.93	1 -32	0.07

Table 3: Class parameters

normal distribution around the expected values with a coefficient of variation (CV) as defined in Table 3. As a job arrives to the system, we will assign it to one of the 8 classes and generate the task parameters accordingly.

The Poisson arrival process will depend on the load of the system we wish to simulate. In our experiments, we vary the load from a medium loaded system to a heavily loaded system to compare performance of our algorithm and the shortest queue greedy algorithm. However, to calculate the load, we need to find the maximum capacity the system can properly handle. The allocation LP presented in the first stage of LoTES is able to calculate an upper bound on such a capacity and so, we will use λ^* found from the allocation LP as our load 1 limit.

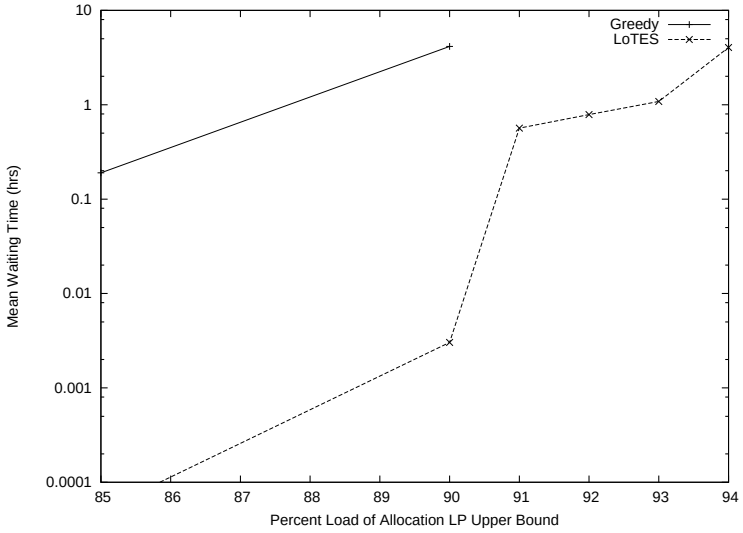

Figure 1: Mean over 20 Runs - Mean Waiting Time Performance

The discrete event simulation is programmed using C++. ILOG CPLEX v12.1 libraries are used as the underlying LP solvers. Figures 1 and 2 show the mean and median performance of LoTES and greedy scheduler

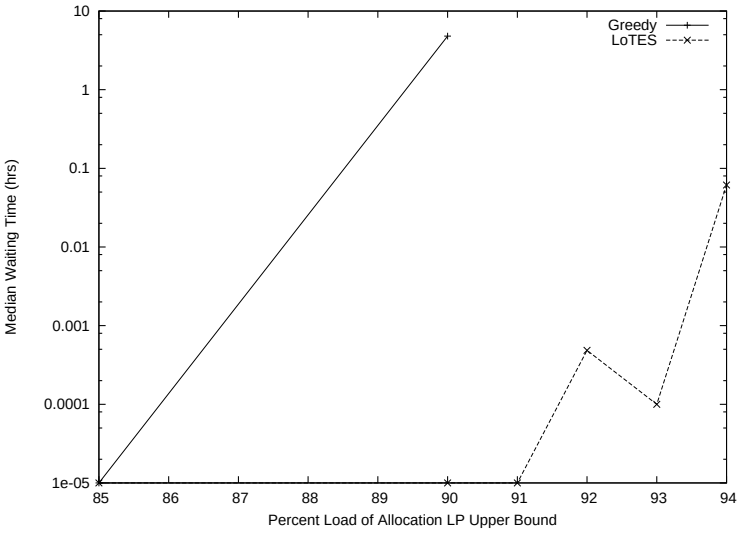

Figure 2: Median over 20 Runs - Mean Waiting Time Performance

respectively across 20 different runs of the parameters shown above.

We initially attempted to simulate 10,000 hours of the system which would represent a data center for just longer than a year. However, the scale of the problem made a complete simulation model difficult to manage as the load of the system increased. Therefore, at the loads above 0.9, the greedy scheduler realized a quickly increasing build up of jobs in the system. As the number of jobs grow, the efficiency of the simulation decreases and simulating 10,000 time units is impossible. Even at 0.85 loads, the greedy scheduler on one of the runs was unable to complete the simulation and could only achieve 7300 hours of simulation after running the code for 5 days. Because of the inability to obtain a complete simulation to 10,000 time units, many of the results are still in a transient state of the system and have not reached steady state.

The LoTES model is able to simulate the full 10,000 time units on a much greater number of runs than the greedy model. It is not until the 0.94 loads that the LoTES model runs into difficulty and cannot reach the proper completion of a simulation. Thus, the actual relative performance difference of the greedy scheduler and LoTES is even greater than that seen in Figures 1 and 2 since the LoTES model has reached a much further point along the simulation and we could reasonably expect the greedy model's performance would further degrade as the simulation time increases.

We know that the system size rapidly grows for the greedy scheduling model at the 0.9 load where the waiting times increase to 4 hours. Figure 3 is an average number of jobs present in the system over all runs of 0.9 loads. It is clear that the greedy scheduler's system size increases rapidly to 750,000 jobs and is still growing after a simulation time of 2000 hours, while the LoTES model maintains a system size of 120,000.

These graphs clearly show that the LoTES model is better adept at handling higher loads than a greedy scheduler which greedily dispatches jobs to the shortest queue. In extreme cases such as 0.9 loads, the greedy scheduler has an expected waiting time of 4 hours while the LoTES model's mean waiting time is only 11 seconds.

The implications of these results is that the LoTES model could improve performance of a data center so that the response time to job requests is significantly decreased. Another advantage to a more efficient allocation

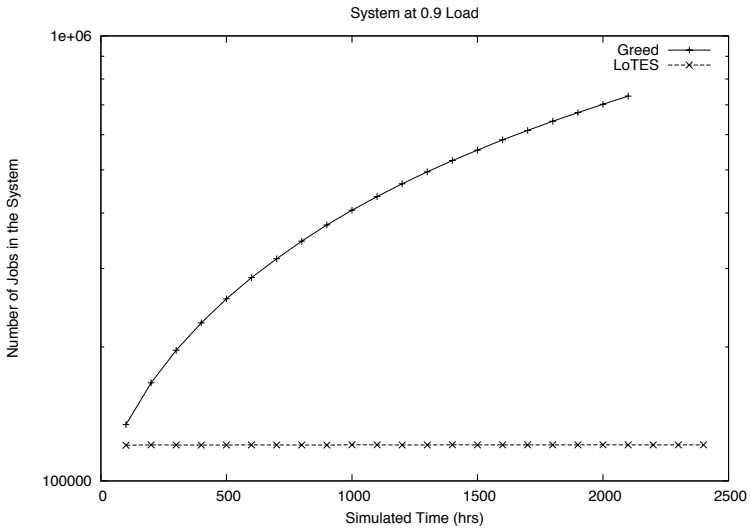

Figure 3: System Size for 0.9 load

scheme is also to maintain performance but reduce the actual required resources. If a scenario where the load is sufficiently low that no waiting times exist for the LoTES scheduler, we could ideally make use of fewer resources. For example, assume that a data center is running at a load of 0.8. If the LoTES model is able to comfortably manage a 0.9 loaded system without incurring a waiting time, the removal of about 10% of the resources could still maintain similar performance. So in a system with 10,000 resources at 0.8 loads, an operator could reduce the system to 88,888 jobs and have a 0.9 load instead. Taking 1,112 servers offline is a significant reduction in overhead costs as well as electrical and cooling costs. However, more realistically, servers are already available in a data center so what is instead possible is to reduce the power to some servers in order to lower electrical and cooling costs while maintaining satisfactory performance through a more optimized scheduling scheme.

Figure 4 provides results on testing the removal of machines from the system. We revisit the system at a load of 0.8. We normalize our results with respect to a lower bound on flow time. Given F, the mean flow time of the system given that a job always arrives to an empty system and can be immediately processed, we present the performance of algorithms by taking the simulated mean flow time and dividing by F. Therefore, a performance of 1 is the best any scheduler can do. As the value increases, the performance of the model degrades. The performance of the Naive scheduler and LoTES model are presented with 100% capacity of machines. Furthermore, the LoTES model with 90% and 95% of machines active is also tested. What can be concluded from the results is that a total reduction of 10% of resources would still lead close to optimal performance if using LoTES. These results are in comparison to the Naive scheduler which witnesses about a 20% increase in flow time while using the full capacity of machines.

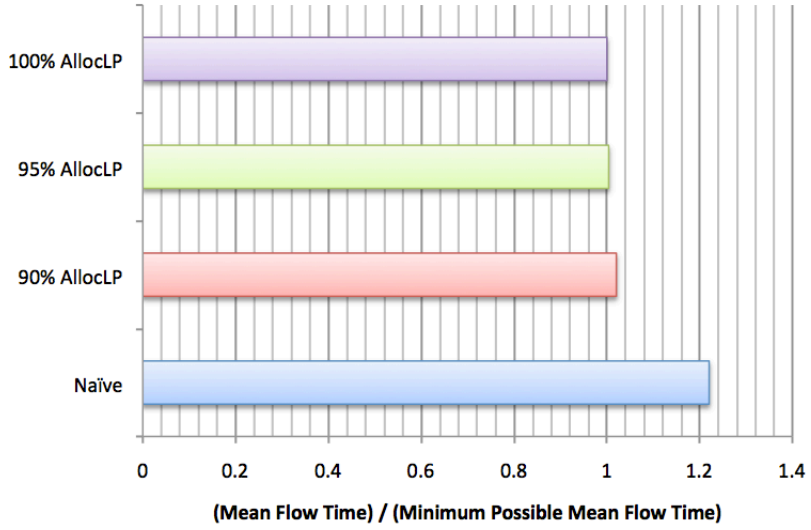

Figure 4: Performance when machines are taken offline.

6 MapReduce Framework

After our initial results, Eran Gabber, had expressed interest in looking at MapReduce jobs. The work until now has been more related to a cloud computing environment than one suitable for MapReduce. MapReduce is a programming model developed at Google to deal with processing and generating large datasets automatically over parallel computers [5]. A commonly used example to explain how a MapReduce job is processed is a word count program; a user program wishes to search through a large set of files and count the number times a certain word appears. To perform such a procedure, MapReduce jobs are comprised of many mapper tasks and reduce tasks. The mapper tasks first process a portion of the files each. These tasks find all the instances of the word and create an intermediate file to be sent to the reduce tasks. Once all mapper tasks have completed their processes, the reduce tasks can then gather the intermediate files to combine the information and create the final output. Following the MapReduce framework, it is possible to quickly and efficiently process a large set of data by breaking the input files down to smaller, more manageable, portions.

From the success of MapReduce, an open-source implementation was created, called Hadoop. The default job scheduler implemented in Hadoop is a FIFO scheduler. However, improvements have been made recently to extend FIFO. Fair sharing [6] is a scheduling plug-in that defines for each user of the system a pool of resources consisting of a certain number of map slots and reduce slots. Each user has access to their own slots, but allows other users access to these slots when idle. Another implementation created to improve data locality is delay scheduling [7]. In delay scheduling, a job is not immediately processed on a machine if the required data is not locally available. To ensure that jobs do not delay too long in an attempt to be processed on a machine with local data, after a pre-determined delay time has elapsed, a queued job will enter service on a machine regardless of locality.

Another scheduling algorithm developed for Hadoop systems was designed to deal with machine heterogeneity [8]. In their work, jobs of different classes arrive to the system and are processed at different speeds depending on the machine giving service. An allocation LP is used to efficiently assign machines to different job classes to take advantage of faster machines whenever possible. The authors show that using more efficient machine-job pairings can lead to significant increase in system throughput while maintaining competitive fairness and locality

objectives when compared to FIFO and fair sharing.

In both works on scheduling of MapReduce environments, the authors look at a system where tasks take up a predefined slot on each machine. Thus, each machine will have a certain number of map slots and reduce slots. That is, if a machine has 8 slots available on it, the scheduling algorithm splits the resources (Processing Cores, Memory, Bandwidth, ...) amongst the 8 tasks that could be scheduled. During the scheduling process, the scheduler must decide for every free slot, which corresponding mapper or reduce task will be serviced. While each task in a job usually requires the same amount of resources, it is not true that the tasks between different jobs are identical as well. Resource usage such as memory requirements for mapper tasks are often specified by the user to determine the size of each task as well as how many tasks are in a job. From our discussion with Eran Gabber, we have been told that resource requirements of jobs can be found to be similar to those presented by Mishra et al. [4] as we used in the previous section. However, further discussions with Doug Down from McMaster University suggests that varying task resource requirements between jobs may not be of interest in the Hadoop scheduling community.

In our work, we assume that tasks of different jobs require different amounts of resources. The resource requirements are made known once the job has arrived to the system. Therefore, rather than predetermined slots on machines, we may schedule jobs on machines as long as there are available resources. Rasooli and Down [8] look at scheduling Hadoop jobs when there is heterogeneity of machines. However, the machines were only different in the speeds at which they can process jobs. We look at machines with differing levels of resource capacities. Specifically, cores and memory requirements. The assumption that tasks of different jobs require different amounts of resources is crucial for the LoTES model. LoTES specifically aims to efficiently match resource requirement profiles of tasks to appropriate machines. However, if one were to assume that each machine has a predetermined number of slots, then LoTES will view all allocations to be equivalent. Hence, if we were to remove the assumption and say all tasks take up a single slot on a machine, the LoTES model does not offer any additional insight.

In the previous system, we were concerned mainly with the response time for jobs. With MapReduce jobs, response time alone can be misleading. Since there are many tasks, the response time to start processing any one task can be very short, but the actual completion of all tasks can take a very long time. Therefore, we are more interested in the performance of the flow time of a system. The flow time is the difference in time from when the job enters the system, and when the last reduce task completes.

Unlike other work on MapReduce systems, we ignore locality as an issue. It is assumed that network bandwidth is much slower than disk bandwidth. Thus, running a task which must access data on a local drive is faster. There are usually two tiers of locality: machine locality and rack locality. Schedulers will often want to process tasks on a local machine where possible. If not, the rack that the data is stored on is a second choice. The further away the rack, the slower the transfer of data will be. However, a recent study showed that the current trend towards increased network bandwidth to equal the level of disk bandwidth will make locality issues irrelevant [9]. This would indicate that a scheduler in the near future could potentially be able to ignore locality problems as well.

To schedule the MapReduce jobs, we propose to use the LoTES model. Instead of single task jobs, we now have hundreds of tasks in each job. The same model can be used, except that the allocation LP must be scaled properly to account for the larger number of tasks per job. However, this process is straightforward as it is possible during the allocation LP step to multiply the resource requirement of a task in a job class with the expected number of tasks in a job. All other steps of LoTES remain the same.

7 Results: MapReduce

In this section, we present the results of simulating the LoTES model with MapReduce jobs. As before, we will use the same machine configurations and job classes. However, each job will now be made of multiple mapper tasks and reduce tasks. Eran Gabber has told us that a typical MapReduce job at Google would contain 200 mapper

tasks and 100 reduce tasks. While this may vary, we assume that all jobs contain exactly these specified number of tasks. Using the data from Table 3, we generate arriving jobs to the system and simulate their traversal through the system using our algorithm.

Figures 5 and 6 show the mean and median performance of LoTES and greedy scheduler respectively across 20 different runs. We present the relative performance of the schedulers to that of the expected flow time if all jobs had entered the system with sufficient resources to immediately start every task as soon as it is available.

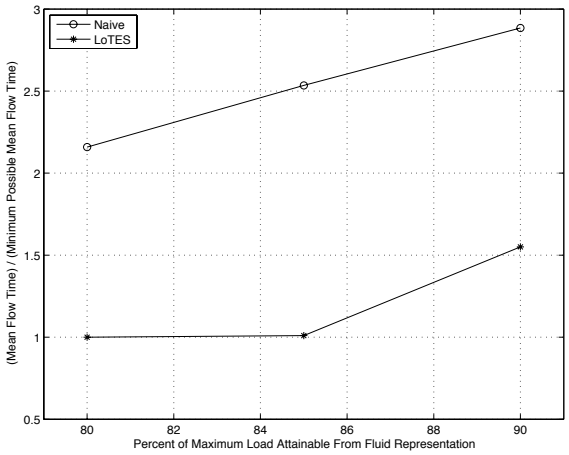

Figure 5: Mean over 20 Runs - Mean Waiting Time Performance

Similar to our non-MapReduce system, the Naive scheduler was unable to complete simulation at higher loads. Therefore, the performance results of the Naive scheduler for 0.85 and 0.90 loads is much less than what the actual performance would be if we could complete the simulation. This leads to the actual relative performance difference of the greedy scheduler and LoTES being even greater than that seen in Figures 5 and 6 since the LoTES model has reached a much further point along the simulation and we could reasonably expect the greedy model's performance would further degrade as the simulation time increases.

The results of the MapReduce system closely follows those of the non-MapReduce environment. One would expect these results since the scheduling of tasks has not changed, only the precedence constraints of some tasks. With a more efficient allocation of tasks onto machines to increase utilization, we are able to boost system throughput and reduce flow time.

8 Future Directions

The results of our simulation show promising benefits from using LoTES. However, more empirical tests need to be performed. In particular, we only examine the system profile of a single Google data center. Mishra et al. [4] examine five different compute clusters and provide job class profiles for each. Their conclusions show that on all data centers, the job classes can always be defined by the eight classes presented in Table 2. It will be necessary to have a more complete set of experiments to test varying the parameters of our system.

Further, the simulation model can be scrutinized because it does not directly use real trace data from a data centre, but aggregate results from Mishra et al.'s [4] paper. Although Google's trace data has been altered to maintain anonymity, we should be able to extract some semblance to a real system and test LoTES using the real

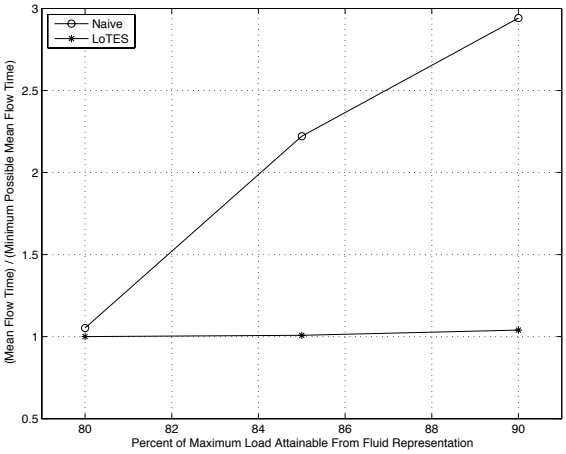

Figure 6: Median over 20 Runs - Mean Waiting Time Performance

data. Further, it would be useful to also test the LoTES model on trace data found form data centers other than those from Google. We would expect the LoTES model to be applicable to a general system with heterogenous machines.

We wish to explore the MapReduce system in more detail. As has already been mentioned, some of the assumptions we've made with our model does not coincide with the current MapReduce community. Most notable of which is the varied resource requirements of tasks. It is known that a common decision of a user is to specify the partition size of their data to be used in a map task. One can think of large data set being split into many 32 MB datasets which are processed individually. Hadoop does have a process for choosing the size to use. One may argue that this size is not fixed and may be changed to suit user needs. However, further investigation is required to understand what is commonly done and whether there is a justifiable reason for our assumption.

We should also further examine locality issues. If it is in fact a significant problem that is likely to continue to exist, we should consider allocations based on locality of data. However, it may not be easily performed since one would require a connection between the location of data and the job classes. It can be that there is no real connection at all and each task, regardless of the job it belongs to, will have data required anywhere on the data center. This may lead to some interesting optimization either from robust allocations to account for locality, or even data storage optimization where a system manager would store data in such a manner that is conducive to ensuring more tasks are served locally.

Finally, an abstraction we maintain in the LoTES model is the resource requirements of tasks in the machine assignment step. The assumption is to use the expected values to define bins. Such an approach can be grossly inaccurate when tasks have stochastic resource requirements. A more advanced model should consider the variance of resource requirements in making bins to obtain more accurate representations.

9 Conclusion

This report detailed current results of testing our LoTES model against a naive scheduler for a typical Google compute cluster with and without MapReduce jobs. We show the potential of using LoTES to efficiently assign resources to jobs in order to realize improved performance as well as efficient usage of limited resources. Given the size of the system, LoTES makes use of tools from queueing theory and optimization to manage a data center.

The queueing analysis provides a high level view of the system where the discreteness of jobs and machines are removed to represent the data center as a fluid. Optimization is then able to use these results to guide the assignment of machines to specific job classes in an intelligent manner.

References

[1] D. Terekhov, T. T. Tran, D. G. Down, and J. C. Beck, "Long-run stability in dynamic scheduling," in *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 2012. To Appear.

[2] D. Terekhov, T. T. Tran, D. G. Down, and J. C. Beck, "Theoretical and algorithmic integration of queueing theory and scheduling." Working Paper, 2012.

[3] S. Andradóttir, H. Ayhan, and D. G. Down, "Dynamic server allocation for queueing networks with flexible servers," *Operations Research*, vol. 51, no. 6, pp. 952–968, 2003.

[4] A. Mishra, J. Hellerstein, W. Cirne, and C. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[7] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "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.

[8] A. Rasooli and D. Down, "An adaptive scheduling algorithm for dynamic heterogeneous hadoop systems," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 30–44, IBM Corp., 2011.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *Proc. USENIX Workshop on Hot Topics in Operating Syst.(HotOS)*, 2011.