Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters

Bikash Sharma *
Pennsylvania State University
University Park 16802
bikash@cse.psu.edu

Victor Chudnovsky
Google Inc.
Seattle 98103
vchudnov@google.com

Joseph L. Hellerstein
Google Inc.
Seattle 98103
jlh@google.com

Rasekh Rifaat
Google Inc.
Seattle 98103
rasekh@google.com

Chita R. Das
Pennsylvania State University
University Park 16802
das@cse.psu.edu

ABSTRACT

Evaluating the performance of large compute clusters requires benchmarks with representative workloads. At Google, performance benchmarks are used to obtain performance metrics such as task scheduling delays and machine resource utilizations to assess changes in application codes, machine configurations, and scheduling algorithms. Existing approaches to workload characterization for high performance computing and grids focus on task resource requirements for CPU, memory, disk, I/O, network, etc. Such resource requirements address *how much* resource is consumed by a task. However, in addition to resource requirements, Google workloads commonly include task placement constraints that determine *which* machine resources are consumed by tasks. Task placement constraints arise because of task dependencies such as those related to hardware architecture and kernel version.

This paper develops methodologies for incorporating task placement constraints and machine properties into performance benchmarks of large compute clusters. Our studies of Google compute clusters show that constraints increase average task scheduling delays by a factor of 2 to 6, which often results in tens of minutes of additional task wait time. To understand why, we extend the concept of resource utilization to include constraints by introducing a new metric, the *Utilization Multiplier (UM)*. UM is the ratio of the resource utilization seen by tasks with a constraint to the average utilization of the resource. UM provides a simple model of the performance impact of constraints in that task scheduling delays increase with UM. Last, we describe how to synthesize representative task constraints and machine properties, and how to incorporate this synthesis into existing performance benchmarks. Using synthetic task constraints and machine properties generated by our methodology, we accurately reproduce performance metrics for benchmarks of Google compute clusters with a discrepancy of only 13% in task scheduling delay and 5% in resource utilization.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Miscellaneous; D.4.8 [**Performance**]: Metrics—*modeling techniques, performance measures.*

General Terms

Performance, modeling, benchmarking.

Keywords

Workload characterization, metrics, benchmarks, performance evaluation.

1. INTRODUCTION

Building compute clusters at Google scale requires having realistic performance benchmarks to evaluate the impact of changes in scheduling algorithms, machine configurations, and application codes. Providing such benchmarks requires constructing workload characterizations that are sufficient to reproduce key performance characteristics of compute clusters. Existing workload characterizations for high performance computing and grids focus on task resource requirements such as CPU, RAM, disk, and network. However, in addition to resource requirements, Google tasks frequently have *task placement constraints* (hereafter, just *constraints*) similar to the Condor ClassAds mechanism [26]. Examples of constraints are restrictions on task placement due to hardware architecture and kernel version. Constraints limit the machines on which a task can run, and this in turn can increase task scheduling delays. This paper develops methodologies that quantify the performance impact of task placement constraints, and applies these methodologies to Google compute clusters. In particular, we develop a methodology for synthesizing task placement constraints and machine properties to provide more realistic performance benchmarks.

Herein, task scheduling refers to the assignment of tasks to machines. We do not consider delays that occur once a task is assigned to a machine (e.g., delays due to operating system schedulers) since our experience is that these delays are much shorter than the delays for machine assignment.

We elaborate on the difference between task resource requirements and task placement constraints. Task resource requirements de-

*This work was done while the author was an intern at Google in summer 2010.

SOCC'11, October 27–28, 2011, Cascais, Portugal.

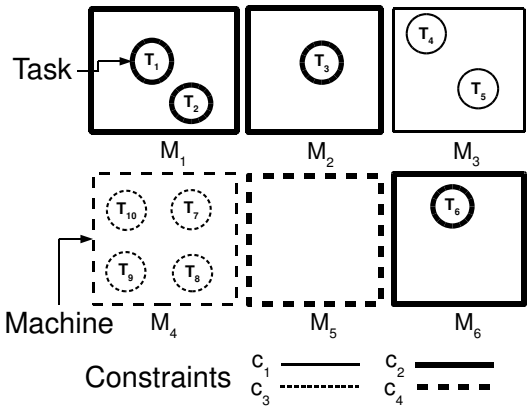


Figure 1: Illustration of the impact of constraints on machine utilization in a compute cluster. Constraints are indicated by a combination of line thickness and style. Tasks can schedule only on machines that have the corresponding line thickness and style.

scribe *how much* resource a task consumes. For example, a task may require 1.2 cores per second, 2.1 GB of RAM per second, and 100 MB of disk space. In contrast, task placement constraints address *which* resources are consumed. For example, a common constraint in Google compute clusters is requiring a particular version of the kernel (e.g., because of task dependencies on particular APIs). This constraint has no impact on the quantities of resource consumed. However, the constraint does affect the machines on which tasks can schedule.

The constraints herein addressed are simple predicates on machine properties. Such constraints can be expressed as a triple of: machine attribute, relational operator, and a constant. An example is "kernel version is greater than 1.2.7".

Why do Google tasks specify constraints? One reason is machine heterogeneity. Machine heterogeneity arises because financial and logistical considerations make it almost impossible to have identical machine configurations in large compute clusters. As a result, there can be incompatibilities between the pre-requisites for running an application and the configuration of some machines in the compute cluster (e.g., kernel version). To address these concerns, Google tasks may request specific hardware architectures and kernel versions. A second reason for task placement constraints is application optimization, such as making CPU/memory/disk tradeoffs that result in tasks preferring specific machine configurations. For these reasons, Google tasks will often request machine configurations with a minimum number of CPUs or disks. A third reason for task constraints is problem avoidance. For example, administrators might use a clock speed constraint for a task that is observed to have errors less frequently if the task avoids machines that have slow clock speeds.

Figure 1 illustrates the impact of constraints on machine utilization in a compute cluster. There are six machines $M_1, \cdots, M_6$ (depicted by squares) and ten tasks $T_1, \cdots, T_{10}$ (depicted by circles). There are four constraints $c_1, \cdots, c_4$. Constraints are indicated by the combinations of line thickness and line styles. In this example, each task requests a single constraint, and each machine satisfies a single constraint. A task can only be assigned to a machine that satisfies its constraint; that is, the line style and thickness of a circle must

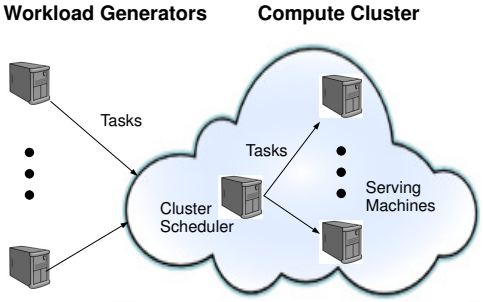


Figure 2: Components of a compute cluster performance benchmark.

be the same as its containing square. One way to quantify machine utilization is the ratio of tasks to machines. In the example, the average machine utilization is $10\,\text{tasks} \div 6\,\text{machines} = 1.66$ tasks per machine. However, tasks with constraint c_3 can be scheduled only on machine M_4 where there are 4 tasks. So, the utilization seen by a newly arriving task that requests c_3 is $4\ \text{tasks} \div 1\,\text{machine} = 4$ tasks per machine. Now consider c_2. There are four tasks that request constraint c_2, and these tasks can run on three machines (M_1, M_2, M_6). So, the average utilization experienced by a newly arriving task that requests c_2 is $4\,\text{tasks} \div 3\,\text{machine} = 1.33$ tasks per machine. In practice, it is more complicated to compute the effect of constraints on resource utilization because: (a) tasks often request multiple constraints; (b) machines commonly satisfy multiple constraints; and (c) machine utilization is a poor way to quantify the effect of constraints in compute clusters with heterogeneous machine configurations.

We use two metrics to quantify the performance impact of task placement constraints. The first metric is *task scheduling delay*, the time that a task waits until it is assigned to a machine that satisfies the task constraints. Task scheduling delay is the primary metric by which performance assessments are done in Google compute clusters because most resources are consumed by tasks that run for weeks or months [24]. An example is a long running search task that alternates between waiting for and processing user search terms. A cluster typically schedules 5 to 10 long-running tasks per hour, but there are bursts in which a hundred or more tasks must be scheduled within minutes. For long-running tasks, metrics such as response time and throughput have little meaning. Instead, the concern is minimizing task scheduling delays when tasks are scheduled initially and when running tasks are rescheduled (e.g., due to machine failures). Our second metric is *machine resource utilization*, the fraction of machine resources that are consumed by scheduled tasks. In general, we want high resource utilizations to achieve a better return on the investment in compute clusters.

Much of our focus is on developing realistic performance benchmarks. As depicted in Figure 2, a benchmark has a workload generation component that generates synthetic tasks that are scheduled by the Cluster Scheduler and executed on Serving Machines. Incorporating task placement constraints into a performance benchmark requires changes to: (a) the Workload Generators to synthesize tasks so that they request representative constraints and (b) the properties of Serving Machines so that they are representative of machines in production compute clusters.

Thus far, our discussion has focused on task placement constraints related to machine properties. However, there are more complex

constraints as well. For example, a job may request that no more than two of its tasks run on the same machine (e.g., for fault tolerance). Although we plan to address the full range of constraints in the future, our initial efforts are more modest. Another justification for our limited scope is that complex constraints are less common in Google workloads. Typically, only 11% of the production jobs use complex constraints. However, approximately 50% of the production jobs have constraints on machine properties.

To the best of our knowledge, this is the first paper to study the performance impact of task placement constraints. It is also the first paper to construct performance benchmarks that incorporate task placement constraints. The specifics of our contributions are best described as answers to a series of related questions.

Q1: Do task placement constraints have a significant impact on task scheduling delays? We answer this question using benchmarks of Google compute clusters. The results indicate that the presence of constraints increases task scheduling delays by a factor of 2 to 6, which often means tens of minutes of additional task wait time.

Q2: Is there a model of constraints that predicts their impact on task scheduling delays? Such a model can provide a systematic approach to re-engineering tasks to reduce scheduling delays and to configuring machines in a cost-effective manner. We argue that task scheduling delays can be explained by extending the concept of resource utilization to include constraints. To this end, we develop a new metric, the **Utilization Multiplier (UM)**. UM is the ratio of the resource utilizations seen by tasks with a constraint to the average utilization of the resource. For example, in Figure 1, the UM for constraint c_3 is $\frac{4}{1.66} = 2.4$ (assuming that there is a single machine resource, machines have identical configurations, and tasks have identical resource demands). As discussed in Section 4, UM provides a simple model of the performance impact of constraints in that task scheduling delays increase with UM.

Q3: How can task placement constraints be incorporated into existing performance benchmarks? We describe how to synthesize representative task constraints and machine properties, and how to incorporate this synthesis into existing performance benchmarks. We find that our approach accurately reproduces performance metrics for benchmarks of Google compute clusters with a discrepancy of only 13% in task scheduling delay and 5% in resource utilization.

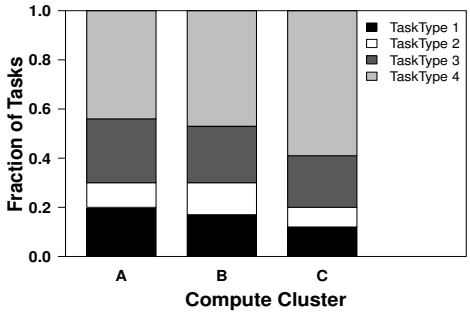


Figure 3: Fraction of tasks by type in Google compute clusters.

The remainder of this paper is organized as follows: Section 2 describes our experimental methodology. Section 3 assesses the impact of constraints on task scheduling delays. Section 4 constructs a simple model of the impact of constraints on task scheduling delays. Section 5 describes how to extend existing performance benchmarks to incorporate constraints. Section 6 discusses related work. Section 7 contains our conclusions and future work.

2. EXPERIMENTAL METHODOLOGY

This section describes the Google task scheduling mechanism and our experimental methodology.

Our experiments use data from three Google compute clusters. We refer to these clusters as A, B and C. The clusters are typical in that: (a) there is no dominant application; (b) there are thousands of machines; and (c) the cluster runs hundreds of thousands of tasks in a day.

There are four task types. Tasks of type 1 are high priority production tasks; tasks of type 4 are low priority, and are not critical to end-user interactions; tasks of type 2 and 3 have characteristics that blend elements of task types 1 and 4. Figure 3 displays the fraction of tasks by type in the Google compute clusters. These fractions are used in Section 5 to construct workloads with representative task placement constraints.

2.1 Google Task Scheduling

Next, we describe how scheduling works in Google compute clusters. Users submit jobs to the Google cluster scheduler. A job describes one or more tasks [24]. The cluster scheduler assigns tasks to machines. A task specifies (possibly implicitly) resource requirements (e.g., CPU, memory, and disk resources). A task may also have task placement constraints (e.g., kernel version).

In principle, scheduling is done in order by task type, and is first-come-first-serve for tasks with the same type. Scheduling a task proceeds as follows:

- determine which machines satisfy the task's constraints,
- compute the subset of machines that also have sufficient free resource capacity to satisfy the task's resource requirements (called the **feasible set**),
- select the "best" machine in the feasible set on which to run the task (assuming that the feasible set is not empty).

To elaborate on the last step, selecting the "best" machine does optimizations such as balancing resource demands across machines and minimizing peak demands within the power distribution infrastructure. Machines notify the scheduler when a job terminates, and machines periodically provide statistics so that the cluster scheduler has current information on machine resource consumption.

2.2 Methodology

We now describe our methodology for conducting empirical studies. A study is two or more experiments whose results are compared to investigate the effects of constraints on task scheduling delays and/or machine resource utilizations. Figure 4 depicts the workflow used in our studies. There are four sub-workflows. The *data preparation sub-workflow* acquires raw trace data from production Google compute clusters. A raw trace is a kind of scheduler checkpoint (e.g., [31]) that contains the history of all scheduling events along with task resource requirements and placement constraints. The *baseline sub-workflow* runs experiments in which there is no modification to the raw trace. This sub-workflow makes use of benchmarks that have been developed for Google compute

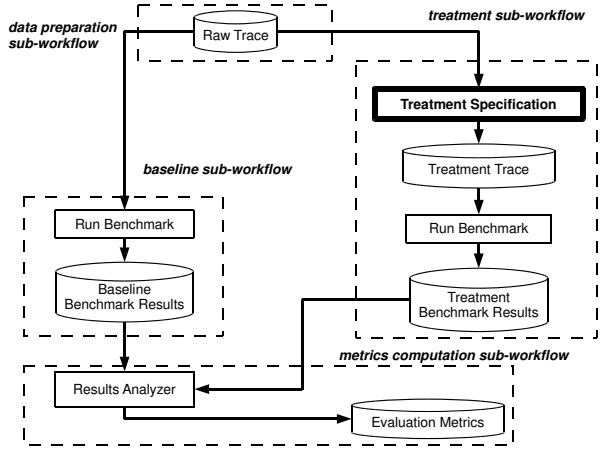


Figure 4: Workflow used for empirical studies. The *Treatment Specification* block is customized to perform different benchmark studies.

clusters. The benchmarks are structured as in Figure 2. Workload Generation is done by synthesizing tasks from traces of Google compute clusters. Then, the real Google cluster scheduler makes scheduling decisions. One version of the benchmark runs with real Serving Machines. In a second version of the benchmark, there are no Serving Machines; instead, the Serving Machines are mocked using trace data to provide statistics of task executions on Serving Machines. Our studies use the latter benchmark for two reasons. First, it is unnecessary to use real Serving Machines. This is because once task assignments are known, task scheduling delays and machine resource utilizations can be accurately estimated from the task execution statistics in the traces. Second, and more importantly, it is cumbersome at best to use real Serving Machines in our studies since evaluating the impact of task constraints requires an ability to modify machine properties. The baseline sub-workflow produces Baseline Benchmark Results.

The *treatment sub-workflow* performs experiments in which machine properties and/or task constraints are modified from those in the raw trace resulting in a Treatment Trace. The block labeled **Treatment Specification** performs the modifications to the raw trace for an experiment. For example, in the next section, the Treatment Specification removes all constraints from tasks in the raw trace. This sub-workflow produces Treatment Benchmark Results.

These computations use the Results Analyzer, which inputs the Baseline Benchmark Results and Treatment Benchmark Results to compute evaluation metrics for task scheduling delays and machine resource utilizations. Our studies employ raw traces from the above mentioned three Google compute clusters. (Although many tasks are scheduled in a day, most consume few resources.) We use a total of 15 raw traces, with 5 traces from each of the three compute clusters. The raw traces are obtained at the same time on successive days during a work week. Because scheduling considerations are more important when resources are scarce, we select traces that have higher resource utilizations.

3. PERFORMANCE IMPACT OF CONSTRAINTS

Short Name	Description	# of values
arch	architecture	2
num_cores	number of cores	8
num_disks	number of disks	21
num_cpus	number of CPUs	8
kernel	kernel version	7
clock_speed	CPU clock speed	19
eth_speed	Ethernet speed	7
platform	Platform family	8

Table 1: Machine attributes that are commonly used in scheduling decisions. The table displays the number of possible values for each attribute in Google compute clusters.

Constraint Names	Constraint Type	Relational Operator
c1.{1}.{1}	arch	=
c2.{1-5}.{1-2}	num_cores	=, $\geq$
c3.{1-3}.{1-2}	max_disks	=, $\geq$
c4.{1-2}.{1-2}	min_disks	=, $\geq$
c5.{1-4}.{1-2}	num_cpus	=, $\geq$
c6.{1-2}.{1}	kernel	=
c7.{1-2}.{1}	clock_speed	=
c8.{1}.{1}	eth_speed	=
c9.{1}.{1}	platform	=

Table 2: Popular task constraints in Google compute clusters. The constraint name encodes the machine attribute, property value, and relational operator.

This section addresses the question **Q1: Do task placement constraints have a significant impact on task scheduling delays?** Answering this question requires considering two factors in combination: (1) the supply of machine resources that satisfy constraints and (2) the resources demanded by tasks requesting constraints.

The constraints satisfied by a machine are determined by the machine's properties. We express machine properties as attribute-value pairs. Table 1 displays machine attributes, the short names of attributes that are used in this paper, and the number of possible values for each machine attribute. To avoid revealing details of Google machine configurations, we do not list the *values* of the machine attributes.

We use Table 1 to infer the number of possible constraints. Recall that a constraint is a triple of machine attribute, relational operator, and value. We only consider constraints that use attribute values of machine properties since constraints that use other values are equivalent to constraints that use values of machine properties. For example, "num_cores > 9" is equivalent to "num_cores > 8" if the maximum value of num_cores is 8. It remains to count the combinations of relational operators and machine properties. For categorical variables, there are two possible relational operators ($\{=, \neq\}$), and for numeric variables there are 6 possible relational operators ($\{=, \neq, \leq, <, \geq, >\}$). Thus, the number of feasible constraints is $\sum_i v_i r_i \approx 400$, where v_i is the number of values of the i-th machine attribute and r_i is the number of relational operators that can be used with the machine attribute.

Not surprisingly, it turns out that only a subset of the possible constraints are used in practice. Table 2 lists the thirty-five constraints

that are commonly requested by tasks. The constraint type refers to a group of constraints with similar semantics. With two exceptions, the constraint type is the same as the machine attribute. The two exceptions are max_disks and min_disks, both of which use the num_disks machine attribute. For the commonly requested constraints, the relational operator is either $=$ or $\geq$. Note that $\geq$ is used with max_disks and min_disks, although the intended semantics is unclear. One explanation is that these are mistakes in job configurations.

The constraint names in Table 2 correspond to the structure of constraints. Our notation is: ***c*** *<**constraint type**>.<**attribute value index**>.<**relational operator index**>*. For example, "c2.4.2" is a num_cores constraint, and so it begins with "c2" since num_cores is the second constraint type listed in Table 2. The "4" specifies the *index* of the value of number of cores used in the constraint (but 4 is not necessarily the value of the attribute that is used in the constraint). The final "2" encodes the $\geq$ relational operator. In general, we encode the relational operators using the indexes 1 and 2 to represent $=$ and $\geq$, respectively.

We provide more insights into the constraints in Table 2. The num_cores constraint requests a number of physical cores, which is often done to ensure sufficient parallelism for application codes. The max_disks constraint requests an upper bound on the number of disks on the machine, typically to avoid being co-located with I/O intensive workloads. The min_disks constraint requests a minimum number of disks on the machine, a common request for I/O intensive applications. The kernel constraint requests a particular kernel version, typically because the application codes depend on certain kernel APIs. The eth_speed constraint requests a network interface of a certain bandwidth, an important consideration for network-intensive applications. The remaining constraints are largely used to identify characteristics of the hardware architecture. The constraints included here are: arch, clock_speed, num_cpus, and platform.

We now describe the supply of machine resources that satisfy constraints. Figure 5 plots the supply of compute cluster CPU, memory, and disk resources on machines that satisfy constraints. The horizontal axis is the constraint using the naming convention in Table 2. Hereafter, we focus on the 21 constraints (of the 35 constraints in Table 2) that are most commonly specified by Google tasks. These constraints are the labels of the x-axis of Figure 5. The vertical axis of that figure is the fraction of the compute cluster resources that satisfy the constraint, with a separate bar for each resource for each constraint. There is much evidence of machine heterogeneity in these data. For example, constraint c3.1.2 is satisfied by machines accounting for 85% of the CPU of compute cluster A, but these machines account for only 60% of the memory of compute cluster A. On the other hand, constraint c6.2.1 is satisfied by machines that account for only 52% of the CPU of compute cluster A but 75% of the memory.

Next, we consider task demands for constraints. Figure 6 displays the demand by task type for compute cluster resources that satisfy the constraints in Table 2. These data are organized by task type. The horizontal axis is the constraint, and the vertical axis is the fraction of the tasks (by type) that request the constraint. Note that it is very common for tasks to request the machine architecture constraint (c1.1.1). This seems strange since from Figure 5 we see that all machines satisfy c1.1.1 in the compute clusters that we study. One reason may be that historically there has been a diversity of machine architectures. Another possible explanation is that the same task may run in other compute clusters, where constraint c1.1.1 does affect scheduling. Other popular constraints are: the number of cores (c2.*.*), the kernel release (c6.*.*), and the CPU clock speed (c7.*.*).

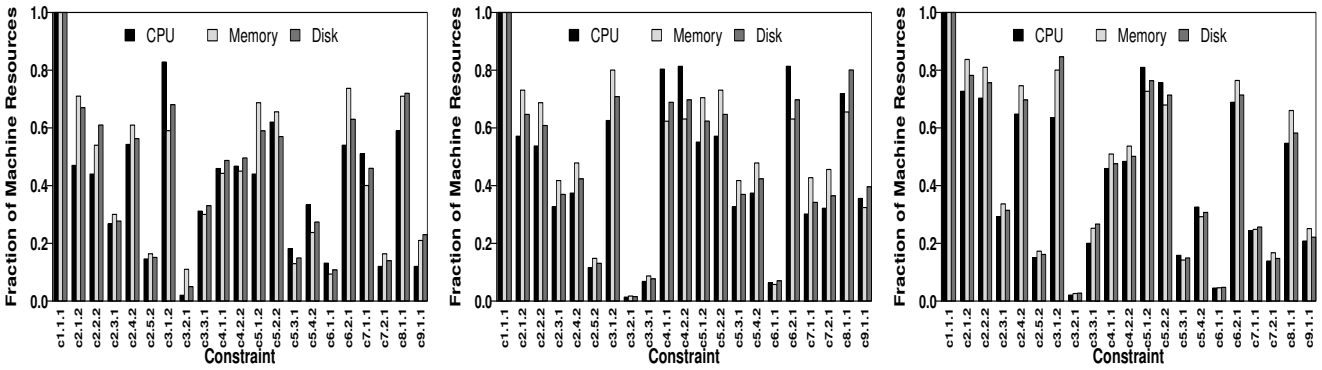
We note in passing that even more insight can be provided by extending Figure 6 to include the resources requested by tasks. However, these are high dimension data, and so they are challenging to present. Further, these data do not provide a complete picture of the impact of constraints on task scheduling delays in that scheduling decisions depend on *all* constraints requested by the task, not just the presence of individual constraints. The constraint characterizations described in Section 5 address this issue in a systematic manner.

Returning to Q1, we assess the impact of constraints on task scheduling delays. Our approach is to have the Treatment Specification in Figure 4 be "remove all constraints". Our evaluation metric is **normalized scheduling delay**, the ratio of the task scheduling delay in the baseline sub-workflow in which constraints are present to the task scheduling delay in the treatment sub-workflow in which all constraints are removed. Thus, a normalized scheduling delay of 1 means that there is no change from the baseline and hence constraints have no impact on task scheduling delays.

Figure 7 plots normalized scheduling delays for the three compute clusters. The horizontal axis is the raw trace file used as input for the experiment (see Figure 4). The vertical axis is the normalized scheduling delay, and there are separate bars for each task type. Observe that **the presence of constraints increases task scheduling delay by a factor of 2 to 6**. In absolute units, this often means tens of minutes of additional task wait time. The reason for this additional wait time is readily explained by examining the supply of machine resources that satisfy constraints and the task demand for these resources. For example, scheduling delays are smaller for tasks that request the first two num_cpus constraints (c5.[1-2].*) compared with tasks that request the clock_speed constraints (c7.*.*). This is because: (a) there are more machine resources that satisfy c5.[1-2].* than those that satisfy c7.*.* (see Figure 5); and (b) the task demand for c7.*.* is much greater than that for c5.[1-2].* (see Figure 6).

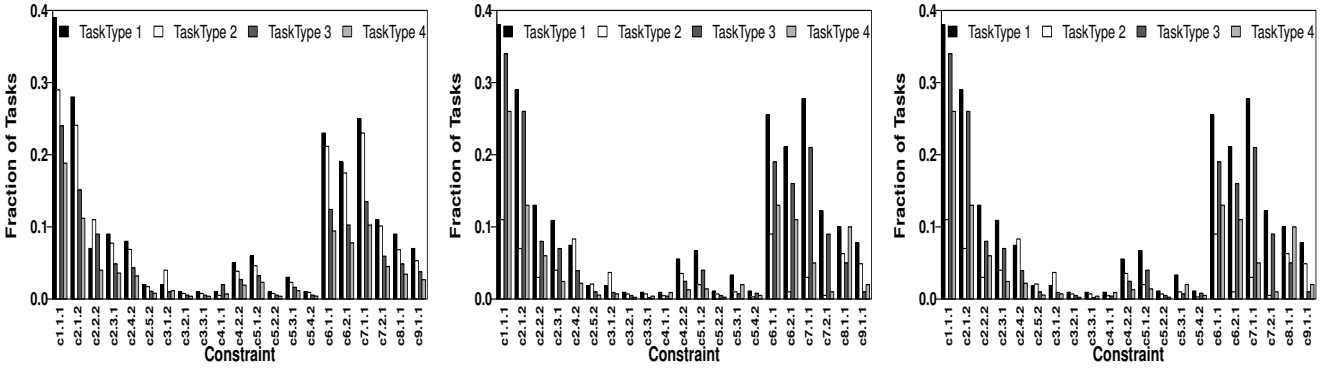
From the foregoing, we conclude that the presence of constraints dramatically increases task scheduling delays in the compute clusters we study. The degree of impact does, however, depend on the load on the compute cluster. Our analysis focuses on periods of heavy load since it is during these times that problems arise. During light loads, constraints may have little impact on task scheduling delays. However, our experience has been that if compute clusters are lightly loaded, then administrators remove machines to reduce costs. This results in much heavier loads on the remaining machines, and hence a much greater impact of constraints on task scheduling delays.

In this and the next section, we do not report results for resource utilizations because our experiments do not reveal significant changes in resource utilizations due to constraints. This is likely because only a small fraction of tasks are unable to schedule due to task placement constraints, and the impact of constraints on utilization is modest. However, the delays encountered by tasks with constraints can be quite large, and so the presence of constraints on task average scheduling delays can be large.



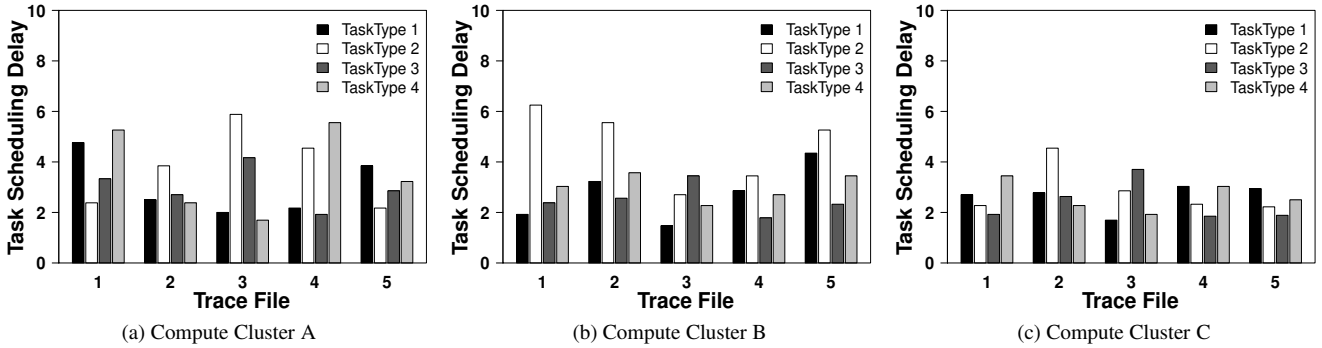
(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 5: Fraction of compute cluster resources on machines that satisfy constraints.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 6: Fraction of tasks that have individual constraints.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 7: Normalized task scheduling delay, the ratio of the task scheduling delays with constraints to task scheduling delays when constraints are removed.

4. MODELING PERFORMANCE WITH CONSTRAINTS

This section provides insight into how constraints impact task scheduling delays. Our approach is motivated by insights from queuing theory [21], in particular, that scheduling delays increase with resource utilizations.

Our initial hypothesis is that task scheduling delays can be explained by average resource utilizations without considering constraints. To test this hypothesis, we conduct studies in which we add to the raw trace 10,000 Type 1 tasks with small resource requirements. We restrict ourselves to TaskType 1 because of their importance. The added tasks have small resource requirements to ensure that they would schedule if there are no task placement constraints. Also, since the tasks have minimal resource requirements, they do not affect resource utilizations; hence, for a single compute cluster, tasks with different constraints see the same average resource utilization. We conduct a total of 315 studies, one study for each of the 15 raw trace files and each of the 21 constraints that we consider. Then, for each constraint c and each compute cluster, we average the delays observed in the five benchmarks of the compute cluster. From this, we calculate normalized task scheduling delays in the same way as is described in Section 3.

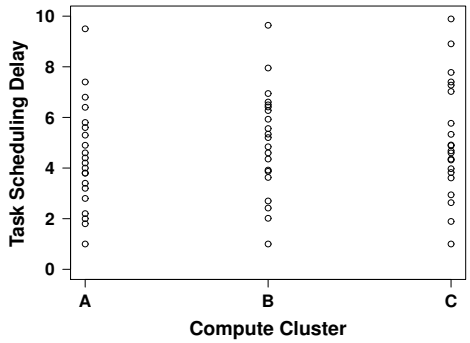


Figure 8: Scheduling delays for tasks with a single constraint. Each point is the average task scheduling delay for 10,000 type 1 tasks with one of the 21 constraints we study.

Figure 8 plots the results for each compute cluster. Recall, that the experiments are structured so that tasks scheduled on a compute cluster see the same average resource utilizations regardless of the task constraint. Thus, if resource utilization without constraints is sufficient to explain task scheduling delays, then the points for a compute cluster should be grouped closely around a single value of task scheduling delay. However, this is not the case. Instead, we see a wide dispersion of points for each of the compute clusters. We conclude that resource utilization by itself cannot explain task scheduling delays if there are task placement constraints.

The foregoing motivates the need to extend the concept of resource utilization to include constraints. We use the term **effective utilization** to refer to the resource utilizations seen by a task with consideration for its constraints. Modeling effective resource utilization allows us to answer question **Q2: "Which constraints most impact task scheduling delays?"** And, answering this question is crucial to determine actions to take to reduce task scheduling delays by modifying applications codes and/or changing machine configurations.

We proceed based on insights obtained from Figure 1. Rather than computing effective utilization directly, we compute the ratio of effective utilization to average utilization. Our metric is the **Utilization Multiplier (UM)**. UM is computed for a particular constraint c and a resource r. In our analysis, r is in the set {CPU, memory, disk}. Computing separate values of UM for each r allows us to address heterogeneous machine configurations.

We construct UM by considering both the task demand imposed by requests for constraint c and the supply of machine resources for which c is satisfied. We begin with task demand. Let d_{cr} be the demand for resource r that is seen by tasks that request constraint c, and let d_r be the total demand for resource r across all tasks. We use the superscript D to indicate a demand metric. So, f^D_{cr} is the fraction of the demand for resource r due to tasks requesting constraint c:

$$f^D_{cr} = \frac{d_{cr}}{d_r}.$$

Next, we analyze the supply of machine resources. Let s_{cr} be the capacity of resource r on machines that satisfy constraint c. (Note that machine capacity is the "raw" supply without consideration for demands from other tasks.) Let s_r denote the total capacity of resource r on all machines. We use the superscript S to denote a supply metric. So, f^S_{cr} is the fraction of total resources r that satisfy constraint c:

$$f^S_{cr} = \frac{s_{cr}}{s_r}.$$

It is more convenient to consider the vector of resources rather than resources individually. Thus, we define for n resources,
$\mathbf{d}_c = (d_{c1}, \cdots, d_{cn})$, $\mathbf{s}_c = (s_{c1}, \cdots, s_{cn})$,
$\mathbf{d} = (d_1, \cdots, d_n)$, $\mathbf{s} = (s_1, \cdots, s_n)$,
$\mathbf{f}^D_c = (f^D_{c1}, \cdots, f^D_{cn})$ and $\mathbf{f}^S_c = (f^S_{c1}, \cdots, f^S_{cn})$.
Note that $\mathbf{f}^S_c, \mathbf{f}^D_c \leq \mathbf{1}$, where $\mathbf{1}$ is the unit vector with dimension n.

UM is the ratio of the fraction of the resource demand for a constraint to its supply. That is:

$$u_{cr} = \frac{f^D_{cr}}{f^S_{cr}}. \quad (1)$$

In vector notation, this is $\mathbf{u}_c = \frac{\mathbf{f}^D_c}{\mathbf{f}^S_c}$ where the division is done element by element. In what follows, all vector operations are done element by element.

$\mathbf{u}_c$ provides a number of insights. First, consider a constraint c that is requested by all tasks and is satisfied by all machines. Then, $\mathbf{f}^S_c = \mathbf{1} = \mathbf{f}^D_c$, and so $\mathbf{u}_c = \mathbf{1}$. That is, if UM is 1, then the constraint has no impact on the resource utilizations seen by tasks with the constraint. In general, we expect that constraints limit the machines on which a task can run. This implies that $\mathbf{f}^S_c < \mathbf{1}$. There are some Google compute clusters in which most resource demands come from tasks that request constraints. For these compute clusters it is likely that for one or more c $\mathbf{f}^D_c > \mathbf{f}^S_c$, and so $\mathbf{u}_c > \mathbf{1}$. That is, in this case, a task requesting c sees larger than average resource utilizations. On the other hand, in some Google compute clusters, tasks request a constraint c that will place the tasks on less desirable (e.g., older) machines thereby reducing effective resource utilization. Such a strategy works if $\mathbf{f}^S_c > \mathbf{f}^D_c$ so that $\mathbf{u}_c < \mathbf{1}$.

$\mathbf{u}_c$ can also be interpreted as an adjusted resource utilization, which motivates our phrase "effective resource utilization." Let ρ_c be the vector of resource utilizations for machines that satisfy constraint c. That is, $\rho_c = \frac{\mathbf{d}_c}{\mathbf{s}_c}$. Let ρ be the vector resource utilizations for all machines, and so $\rho = \frac{\mathbf{d}}{\mathbf{s}}$. So,

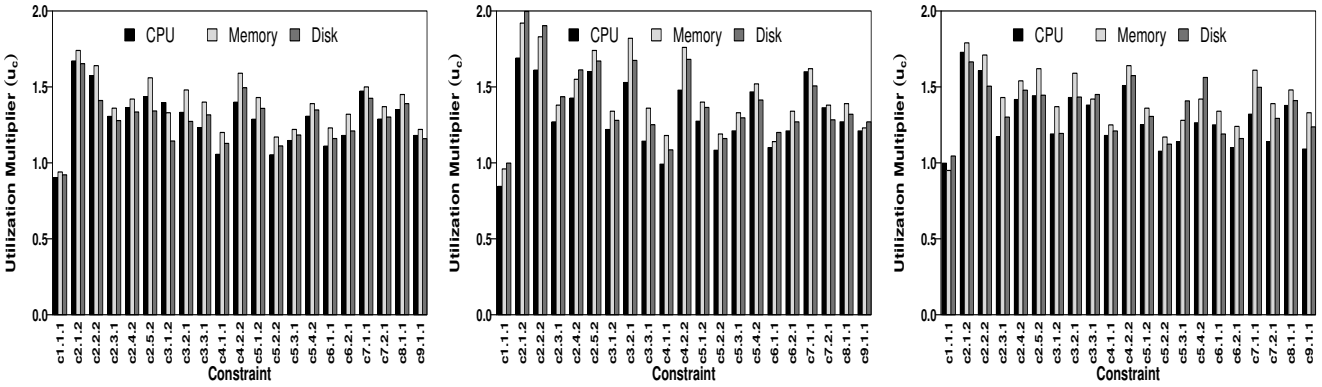
$$\mathbf{u}_c \quad = \quad \frac{\mathbf{f}^D_c}{\mathbf{f}^S_c} = \frac{\mathbf{d}_c}{\mathbf{d}}\frac{\mathbf{s}}{\mathbf{s}_c} = \frac{\mathbf{d}_c}{\mathbf{s}_c}\frac{\mathbf{s}}{\mathbf{d}} = \frac{\rho_c}{\rho}.$$

That is, the utilization of resource r seen by tasks that request constraint c is a factor of u_{cr} larger than the average utilization of resource r.

Often, task scheduling delays are determined by the bottleneck resource rather than the entire resource vector. The bottleneck resource is the resource that has the largest utilization. Thus, we define the **maximum UM**, denoted by u^*_c, to be the scalar

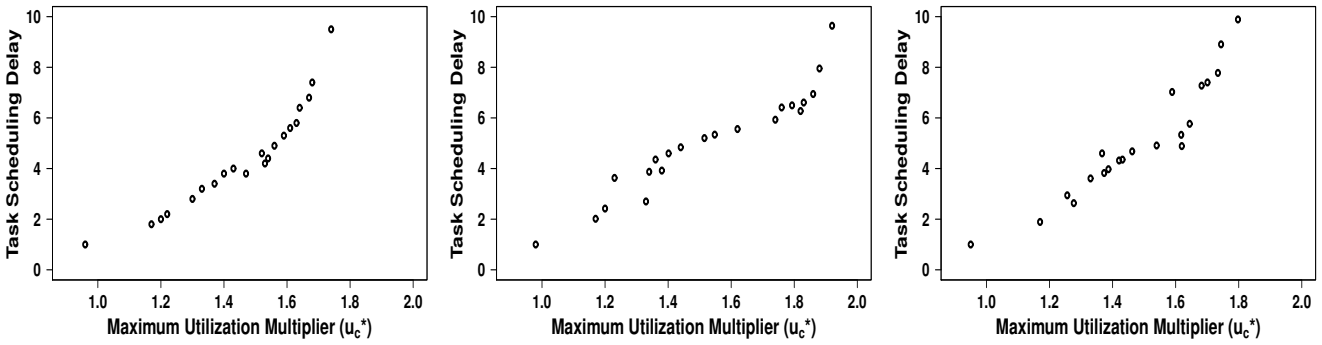
$$u^*_c = max_r(u_{cr}). \quad (2)$$

Figure 9 displays $\mathbf{u}_c$ by CPU, memory, and disk for Type 1 tasks. Although the values of $\mathbf{u}_c$ vary from compute cluster to cluster, there is a general consistency in the relative magnitude of $\mathbf{u}_c$. For example, $\mathbf{u}_{c.1.1.1}$ is consistently smaller than the other constraints, and $\mathbf{u}_{c.2.1.2}$ is consistently one of the largest values. For the most part, $\mathbf{u}_c > \mathbf{1}$. The one exception is c1.1.1 where there are a few instances in which $\mathbf{u}_c \approx 1$. At a first glance, this is surprising since



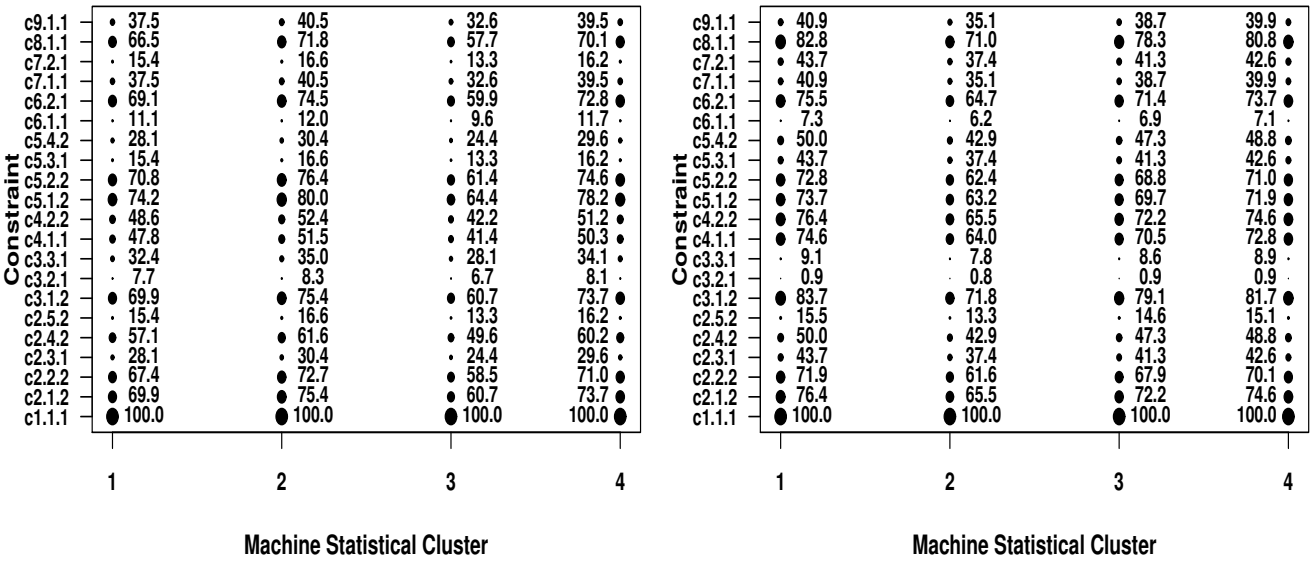
(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 9: Utilization Multiplier by resource for constraints.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

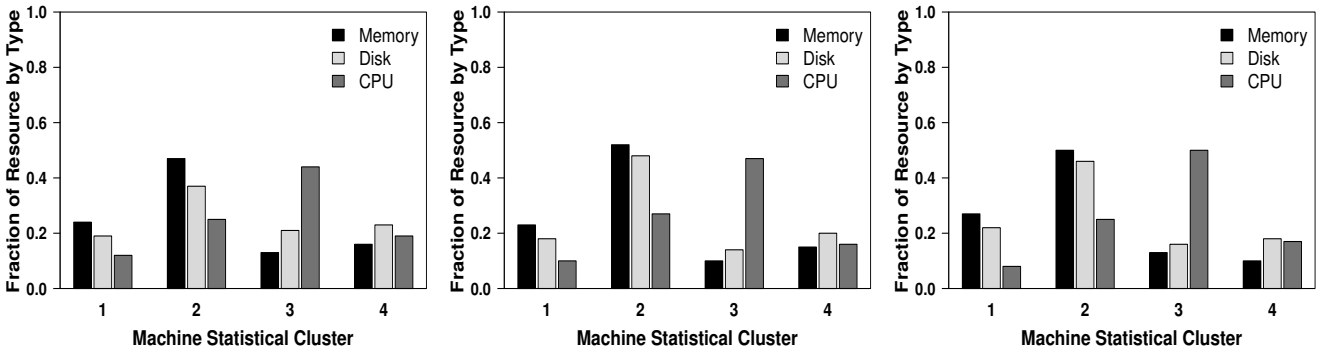
Figure 10: Impact of maximum utilization multiplier on normalized task scheduling delay.



(a) Compute Cluster A (b) Compute Cluster B

Figure 11: Constraint frequency vectors for machine statistical clusters for compute clusters A and B. The numbers are the percentage of machines in the statistical cluster that satisfy the constraint.

Figure 6 shows that a large fraction of tasks request c1.1.1. However, UM is the ratio of the demand for resources with *c* to the supply of these resources. From Figure 5, we know that there is a large



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 12: Resource distribution for machine statistical clusters.

supply of machine resources that satisfy c1.1.1. Hence, $\mathbf{u}_{c.1.1.1}$ is small.

We return to the experiment described earlier in this section to see if UM provides a better explanation of task scheduling delays than average resource utilization. Figure 10 displays the relationship between u_c^* and normalized task scheduling delay. Note that for the most part, **UM provides a simple model of the performance impact of constraints in that task scheduling delays increase with maximum UM (u_c^*)**. The relationship is linear for smaller values of u_c^*, but increases faster than linearly for larger u_c^*. Such curves are common in queuing analysis [21].

Figure 10 indicates that if we order constraints by UM, then we have also ordered constraints by their impact on task scheduling delays. From Figure 9, we see that in all three compute clusters, the constraint with the largest UM (and hence the largest impact on task scheduling delay) is c2.1.2, the first num_cores constraint. For compute cluster A, the constraints with the second and third largest impact on scheduling delays are clock_speed (c7.1.1) and min_disks (c4.2.2) constraints respectively. The situation is a bit different for compute cluster B. Here, the second and third largest scheduling delays are due to max_disks (c3.2.1) and min_disks (c4.2.2) constraints.

Much insight can be obtained from the simple model that task scheduling delays increase with UM. For example, consider the problem of resolving large scheduling delays for a task that has small resource requirements and many task placement constraints. We first compute the UM of the task's constraints in the compute cluster in which the task has long scheduling delays. We focus on the constraints with the largest u_c^*. It may be that some of these constraints can be eliminated from the task's request, especially if the constraints are actually preferences rather than requirements. However, if constraints with large u_c^* cannot be eliminated, an alternative is to find another compute cluster in which u_c^* is smaller. Such a cluster will either have less demand for machine resources that satisfy c or a larger supply of such resources.

5. BENCHMARKING WITH CONSTRAINTS

This section investigates how to synthesize constraints to produce representative performance benchmarks. There are two parts to this: (1) characterizing task constraints and machine properties; and (2) incorporating synthetic constraints and properties into performance benchmarks.

5.1 Constraint Characterization

To characterize task placement constraints, we must address both tasks and machines. We use statistical clustering[1] to construct groups of tasks and machines, a commonly used approach in workload characterization [4]. A **task statistical cluster** is a group of tasks that are similar in terms of constraints that the tasks request. A **machine statistical cluster** is a group of machines that are similar in terms of constraints that the machines satisfy. Tasks and machines belong to exactly one statistical cluster.

Task and machine statistical clusters have a common structure, and so the following definitions apply to both. To this end, we use the term entity to refer to both task and machine. A statistical cluster has two metrics. The first metric is the scalar **cluster occurrence fraction**. This is the fraction of all entities (of the same type in the same compute cluster) that are members of the statistical cluster. The second metric is the **constraint frequency vector**. This vector has one element for each constraint c. The vector element for c is the scalar **constraint occurrence fraction**, which is the fraction of entities in the statistical cluster that "have" the constraint c. For tasks, "have" means that the task requests the constraint, and for machines "have" means that the machine satisfies the constraint.

We use the term **machine constraint characterization** to refer to the set of machine statistical clusters. The **task constraint characterization** is defined analogously. A **constraint characterization** is the combination of the machine constraint characterization and the task constraint characterization.

We construct machine statistical clusters by using a binary feature vector. For each machine, the i-th element of its feature vector is 1 if the machine satisfies the i-th constraint; otherwise, the element is 0. The elements of the feature vector are indexed by descending value of the maximum UM of the constraint. That is, $u_{c_i}^* \geq u_{c_{i+1}}^*$. Initially, we used k-means [17] to construct machine statistical clusters. However, this proved to be unnecessarily complex for our data. Rather, it turns out that a simple approach based on sorting works extremely well. Our approach starts by sorting the machine feature vectors lexicographically. We form machine statistical clusters by placing together machines that are adjacent in the sorted sequence as long as no element in their constraint frequency vector differs by more than 0.05. The choice of 0.05 is empirical; it is based on the trade-off between the goals of having few statistical

[1] Unfortunately, the statistics and cloud computing communities both use the term cluster but with very different semantics. We distinguish between these semantics by using the phrases "statistical cluster" and "compute cluster".

clusters and having statistical clusters with homogeneous machine properties. The result is four machine statistical clusters for each of the three compute clusters that we study.

For task statistical clusters, we also use a binary feature vector. Here, element i is 1 if the task requests constraint c_i; otherwise, the vector value is 0. Task statistical clusters are constructed using k-means [17] on the task feature vector. K-means is used to construct task statistical clusters because many values in the task constraint frequency vectors are close to 0.5 and this precludes using simple clustering algorithms as we did with constructing machine statistical clusters.

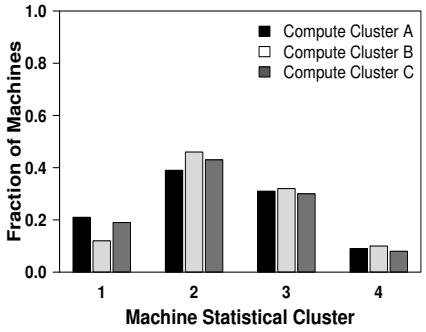


Figure 13: Occurrence fraction for machine statistical clusters.

Algorithm 1 : Extend an existing benchmark to include task placement constraints.

Require: Existing benchmark that generates machine resource capacities and task resource requirements.
1: **if** event = benchmark initialization **then**
2: Assign properties to machines using Algorithm 2.
3: **end if**
4: **if** event = task arrival **then**
5: Assign constraints to the task using Algorithm 3.
6: **end if**
7: **if** event = task is a candidate to run on a machine **then**
8: Determine if the machine satisfies the constraints required by the task using Algorithm 4.
9: **end if**

Figure 11 displays the constraint frequency vector for machine statistical clusters A and B. The results for compute cluster C are similar to those for A and B. More complete data can be found in [28]. The horizontal axis is the machine statistical cluster, and the vertical axis is the constraint. Each point is sized in proportion to the fraction of machines in the statistical cluster that satisfy the constraint (with the actual value marked alongside). For example, in Figure 11(a), about 74.2% of the machines that belong to statistical cluster 1 satisfy constraint c5.1.2. We see that the machine statistical clusters are fairly consistent between the compute clusters. Indeed, even though there are thousands of machines in each compute cluster, the variation in machine properties can largely be explained by four statistical clusters.

Figure 12 displays the fraction of resources supplied by the machine statistical clusters. The horizontal axis is the machine statistical cluster, and the vertical axis is the fraction of resources by resource type. For instance, in Figure 12 (a), approximately 25% of the memory capacity in compute cluster A is on machines belonging to statistical cluster 1.

Algorithm 2 : Assign properties to a machine.

Require: MachineOccurrenceFraction (Figure 13), MachineConstraintFrequencyVector (Figure 11), Machine
1: cluster = randomly choose cluster weighted by MachineOccurrenceFraction
2: mcfv = MachineConstraintFrequencyVector for cluster
3: **for** constraint in mcfv **do**
4: **if** random(0,1) $\leq$ mcfv[constraint].ConstraintOccurrenceFraction **then**
5: Machine.add(property that satisfies the constraint)
6: **end if**
7: **end for**

Algorithm 3 : Assign task placement constraints to a task.

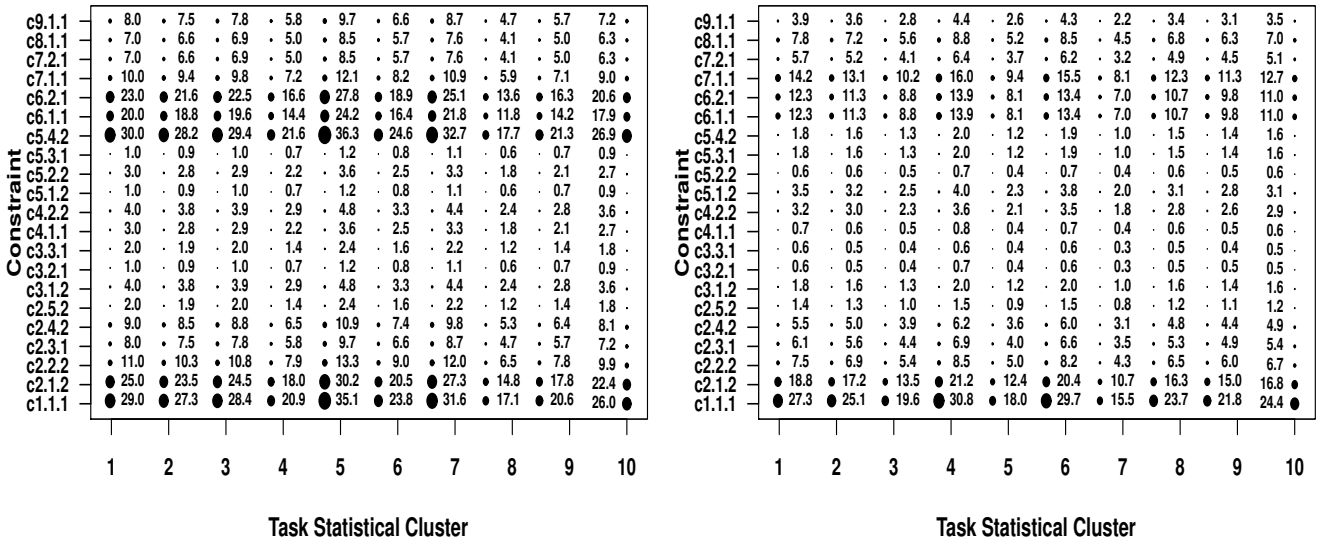
Require: TaskOccurrenceFraction(Figure15), TaskConstraintFrequencyVector(Figure14), Task
1: cluster = randomly choose cluster weighted by TaskOccurrenceFraction
2: tcfv = TaskConstraintFrequencyVector for cluster
3: **for** constraint in tcfv **do**
4: **if** random(0,1) $\leq$ tcfv[constraint].ConstraintOccurrenceFraction **then**
5: Task.add(constraint)
6: **end if**
7: **end for**

Figure 13 shows the occurrence fractions of the machine statistical clusters in the three Google compute clusters. The horizontal axis is the machine statistical cluster and the vertical axis is the fraction of machines that belong to the statistical cluster. For example, in Figure 13, approximately 20% of the machines in compute cluster A belong to machine statistical cluster 1.

Figure 14 displays the constraint frequency vectors of the task statistical clusters for compute clusters A and B. The figure is structured in the same way as for the machine statistical clusters in Figure 11. The horizontal axis is the task statistical cluster; the vertical axis is the constraint; each point is sized in proportion to the fraction of tasks in the statistical cluster that request the constraint (with the actual value marked alongside). The task statistical clusters are more difficult to interpret than the machine statistical clusters because: (a) many constraints have values closer to 0.5; (b) there is more similarity between the statistical clusters; and (c) it is difficult to relate task clusters in one compute cluster to task clusters in another compute cluster. Figure 15 displays the occurrence fractions of the task statistical clusters by task type. The horizontal axis is the task statistical cluster, and the vertical axis is the fraction of tasks (by type) that belong to the statistical cluster. For instance, in Figure 15 (a), approximately 4% of Type 1 tasks belong to task statistical cluster 1. To provide further insight into the nature of tasks in clusters, Figure 16 shows the distribution of resources demanded (by task type) for the 10 task statistical clusters. These data can be used in performance benchmarks to specify resource requirements that are representative of Google compute clusters.

5.2 Extending Performance Benchmarks

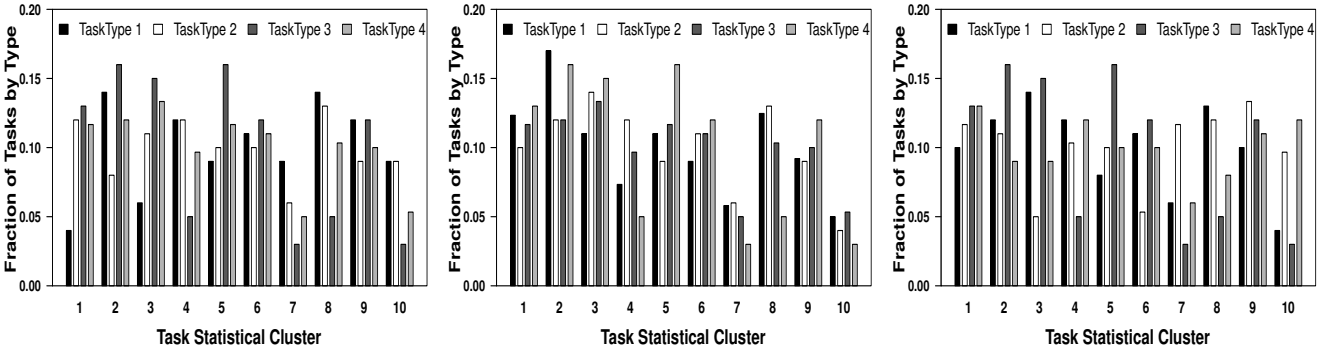
Next, we show how to extend existing performance benchmarks to incorporate task constraints and machine properties. First, we detail how to characterize and synthesize representative task constraints and machine properties. Then, we show that how to incor-



(a) Compute Cluster A

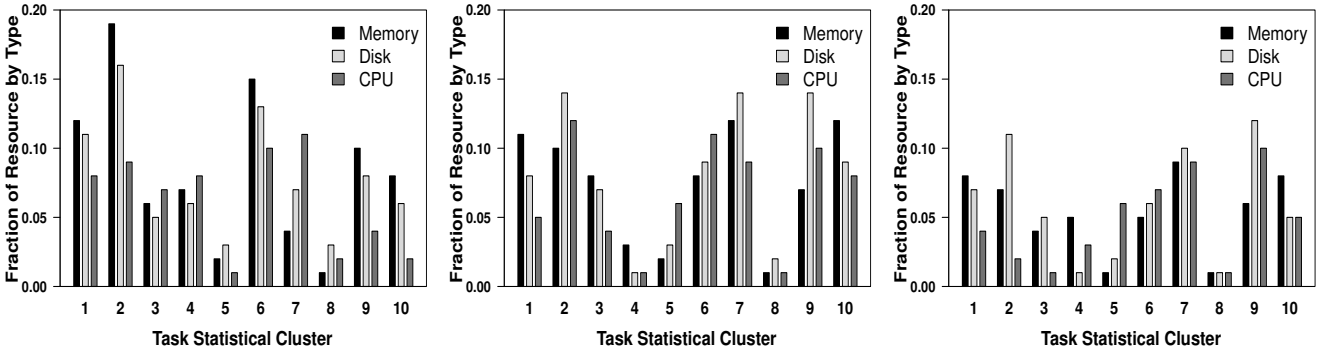
(b) Compute Cluster B

Figure 14: Constraint frequency vectors for task statistical clusters for compute clusters A and B. The numbers are the percentage of tasks in the statistical cluster that request the constraint.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 15: Occurrence fraction for task statistical clusters.



(a) Compute Cluster A (b) Compute Cluster B (c) Compute Cluster C

Figure 16: Resource distribution for task statistical clusters.

porate this synthesis into existing performance benchmarks such as those described in [13, 16].

Algorithm 1 describes the logic added to such an existing bench-

Algorithm 4 : Determine if a task can be run on a machine. Returns TRUE if Task can run on Machine.

```
Require: Task, Machine
1: for constraint in Task.constraints() do
2:     if constraint is not in Machine.constraints() then
3:         return FALSE
4:     end if
5: end for
6: return TRUE
```

mark in order to incorporate representative constraints. This logic is event-based, with processing done when the following events occur: (a) the benchmark is initialized; (b) a task arrives; and (c) a task is a candidate to be placed on a machine.

Algorithm 2 details how to assign properties to a machine. In step 1, a machine statistical cluster is chosen randomly based on the occurrence fractions of machine statistical clusters (e.g., Figure 13). The algorithm uses the constraint frequency vector of the chosen machine statistical cluster (e.g., Figure 11) and a random number generator to select constraints that the machine must satisfy. The structure of constraints in Figure 2 makes it trivial to specify a property that satisfies a constraint. For example, if the constraint is "num_cores $\geq$ 2", then the machine is assigned the value of 2 for its num_cores attribute. In general, there can be logical inconsistencies in the properties inferred in this manner. However, this problem does not arise for the set of constraints of the four machine statistical clusters for the compute clusters that we study.

Algorithm 3 assigns constraints to tasks in a manner similar to what Algorithm 2 does for machines. A task statistical cluster is chosen randomly based on the occurrence fractions of task statistical clusters (e.g., Figure 15). Then, the constraint frequency vector of the chosen task statistical cluster (e.g., Figure 14) is used to assign constraints. Note that the logic assumes that the task type is known. If this is not the case, a task type can be chosen randomly using the distributions in Figure 3.

Algorithm 4 describes the logic added to an existing compute cluster scheduler to take into account task constraints. Before a task is assigned to a machine, the cluster scheduler calls Algorithm 4. The algorithm returns true only if the machine satisfies all of the constraints required by the task.

How accurately do the foregoing algorithms reproduce the performance characteristics of Google compute clusters? To answer this question, we construct experiments by changing the Treatment Specification block in Figure 4 to synthesize constraints for machines and tasks using Algorithm 1. Specifically, for all machines in the trace, we remove the machine's properties and then apply Algorithm 2 to generate synthetic constraints that in turn determine the properties that are assigned to the machines. Similarly, for all tasks in the trace, we remove its constraints and then apply Algorithm 3 to generate synthetic constraints that replace the task's constraints in the raw trace. Each study produces two metrics: average error in task scheduling delay and average error in machine resource utilization. The error in task scheduling delays is computed as the percent deviation of the average task scheduling delay in the treatment (synthesized constraints) from the average scheduling delay in the baseline (constraints in the raw trace). The error in compute cluster resource utilization is computed similarly.

Figure 17 shows the results of our evaluation of the workload characterizations for task scheduling delays. Figure 17(a) displays the errors introduced if we use synthetic constraints for tasks and the actual machine properties. We see that synthesizing task constraints introduces an error of approximately 8%. Figure 17(b) displays the errors in task scheduling delays if only machines constraints are synthesized. Here, we see an average error of around 5%. Figure 17(c) displays the errors if both task and machine constraints are synthesized. The average error is approximately 13%.

Figure 18 analyzes the errors in compute cluster resource (CPU, memory and disk) utilization introduced by using synthetically generated constraints. Figure 18(a) displays the results when we synthesize task constraints and use the actual machine properties. We see that this introduces an error of around 6%. Figure 18(b) shows the errors resulting from synthesizing machines constraints and using the actual tasks constraints. Here, we see an average error of around 3%. In Figure 18(c), we see that the average error is approximately 5% if both task and machine constraints are synthesized.

From these results, we conclude that **our constraint synthesis produces representative workloads for Google compute clusters**. Specifically, synthetic workloads generated using our constraint characterizations result in task scheduling delays that differ by an average of 13% from what is produced by using the production machines/tasks. And, our approach produces machine resource utilizations that differ by an average of 5% from the production machines/tasks.

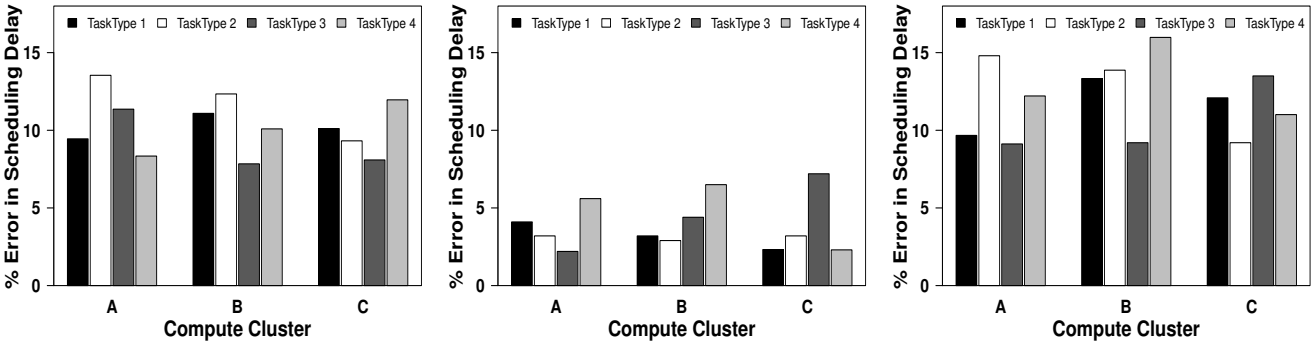
Although our methodology for building performance benchmarks that incorporate constraints is illustrated using data from Google compute clusters, the methodology is not specific to Google. Our characterization of constraints for machines and tasks is done in a general way using clustering, cluster occurrence fractions, and cluster constraint frequency vectors. Our approach to incorporating these characterizations into performance benchmarks is also quite general, as described in Algorithm 1-Algorithm 4.

6. RELATED WORK

Many compute clusters incorporate task placement constraints. Examples include: the Condor system [10] that uses the ClassAds mechanism [31], IBM's load balancer [18], the Utopia system [32], and grid toolkits [15].

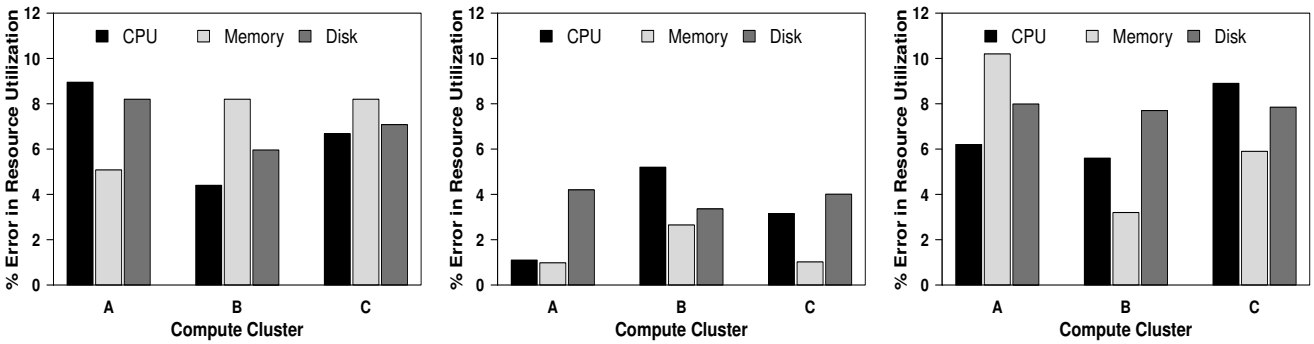
Other related work includes predicting the queuing delay [29], advance reservation and queue bounds estimation [3, 25] for jobs in Grid computing and parallel supercomputers context. Although these studies address the performance impact of resource requirements, they do not consider task placement constraints. A central part of our contribution is the characterization of task placement constraints in terms of task and machine statistical clusters and their properties (e.g., occurrence fractions and frequency vectors). Further, we introduce a new metric, Utilization Multiplier (UM), that extends resource utilization to include constraints. UM provides a simple model of task scheduling delays in the presence of constraints in that task scheduling delays increase with UM.

There is a vast literature on workload characterization for distributed systems in general and compute clouds in particular. Examples include: web server workload characterization [1,2,14], scientific and high performance computing workloads [5, 9, 23], chip-level characteristics [22, 27], resource consumption of Google compute clus-



(a) Synthetic Task Constraints (b) Synthetic Machine Properties (c) Synthetic Constraints and Properties

Figure 17: Percent error in task scheduling delay resulting from using synthetic task constraints and/or machine properties.



(a) Synthetic Task Constraints (b) Synthetic Machine Properties (c) Synthetic Constraints and Properties

Figure 18: Percent error in compute cluster resource utilizations resulting from using synthetic task constraints and/or machine properties.

ters [6,24], and characterizations of MapReduce [7,12,19]. Further, there has been work in the area of workload generation and benchmarking including: MapReduce workload generation [8, 16, 20], Web 1.0 benchmark tools such as *ab*, *httpperf*, Web 2.0 benchmarks [30], and the Yahoo! Cloud Servicing Benchmark [11]. However, none of the foregoing provide characterizations of task placement constraints and machine properties, nor does the existing literature consider how to incorporate constraints into performance benchmarks.

7. CONCLUSIONS AND FUTURE WORK

There has been much prior work on task scheduling that considers resource requirements that address *how much* resource tasks consume. This paper addresses the performance impact of task placement constraints. Task placement constraints impact *which* resources tasks consume. Task placement constraints, such as characteristics specified by the Condor ClassAds mechanism, provide a way to deal with machine heterogeneity and diverse software requirements in compute clusters. Our experience at Google suggests that task placement constraints can have a large impact on task scheduling delays.

This paper is the first to develop a methodology that addresses the performance impact of task placement constraints. We show that in Google compute clusters, constraints can increase average task scheduling delays by a factor of 2 to 6, which often means tens of minutes of additional task wait time. To understand why, we introduce a new metric, the Utilization Multiplier (UM), that extends the concept of resource utilization to include constraints. We show that task scheduling delays increase with UM for the tasks that we study. We also show how to characterize and generate representative task constraints and machine properties, and how to incorporate synthetic constraints and properties into existing performance benchmarks. Applying our approach to Google compute clusters, we find that our constraint characterizations accurately reproduce production performance characteristics. Specifically, our constraint synthesis produces benchmark results that differ from production workloads by an average of 13% in task scheduling delays and by an average of 5% in machine resource utilizations.

Although our data is obtained from Google compute clusters, the methodology that we develop is general. In particular, the UM metric applies to any compute cluster that employs a ClassAds style of constraint mechanism. We look forward to seeing data that provides insights into the performance impact of task placement constraints in other compute clusters.

We are pursuing two extensions to this work. The first is to address constraints that are preferences rather than requirements. For example, in some situations, a task may prefer to run on a machine with 4 cores, but the task may not require this. We expect that UM will be a useful tool in these studies since it allows us to understand effective task utilizations with and without satisfying a preferential constraint. A second extension is to study constraints that apply to collections of tasks. For example, there may be a requirement that tasks be assigned to the same machine because of shared data. Characterizing and benchmarking with inter-task constraints

is complicated because tasks and machines cannot be addressed in isolation.

Acknowledgements

We wish to thank Google Inc. for providing the computing infrastructure that supported this research. And, our sincere thanks to the many Google engineers who provided valuable feedback on this research. This work is supported in part by National Science Foundation (NSF) grant No. CNS-0721479 and a research grant from Google Inc.

8. REFERENCES

[1] M. F. Arlitt and C. L. Williamson. Web server workload characterization: the search for invariants. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1996.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1998.

[3] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006.

[4] M. Calzarossa and D. Ferrari. A sensitivity study of the clustering approach to workload modeling. *SIGMETRICS Performance Evaluation Review*, 1986.

[5] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, 1999.

[6] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. Analysis and lessons from a publicly available google cluster trace. Technical Report UCB/EECS-2010-95, UC Berkeley, 2010.

[7] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. A methodology for understanding mapreduce performance under diverse workloads. Technical Report UCB/EECS-2010-135, UC Berkeley, 2010.

[8] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. Towards understanding cloud performance tradeoffs using statistical workload analysis and replay. Technical Report UCB/EECS-2010-81, UC Berkeley, 2010.

[9] W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *Proceedings of the IEEE Workload Characterization Workshop*, 2001.

[10] Condor Project. University of Wisconsin, . `http://www.cs.wisc.edu/condor`.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[13] Y. Denneulin, E. Romagnoli, and D. Trystram. A synthetic workload generator for cluster computing. *Parallel and Distributed Processing Symposium*, 2004.

[14] D. Ersoz, M. S. Yousif, and C. R. Das. Characterizing network traffic in a cluster-based, multi-tier data center. *Distributed Computing Systems, International Conference on*, 2007.

[15] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1996.

[16] A. S. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson. Statistics-driven workload modeling for the cloud. Technical Report UCB/EECS-2009-160, UC Berkeley, 2009.

[17] J. A. Hartigan and M. A. Wong. A K-means clustering algorithm. *Applied Statistics*, 1979.

[18] IBM. IBM LoadLeveler, . `http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf`.

[19] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. *IEEE Cluster Computing and the Grid*, 2010.

[20] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom. Mrbench: A benchmark for mapreduce framework. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, 2008.

[21] L. Kleinrock. *Queueing Systems*. Wiley-Interscience, 2nd edition, 1975.

[22] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, 2008.

[23] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel Distributed Computing*, 2003.

[24] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *SIGMETRICS Performance Evaluation Review*, 2009.

[25] D. Nurmi, J. Brevik, and R. Wolski. Qbets: Queue bounds estimation from time series. In *In Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2007.

[26] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.

[27] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS, 2009.

[28] B. Sharma, V. Chudnovsky, H. J. L., R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. Technical Report CSE#11-005, CSE Dept., Pennsylvania State University, 2011.

[29] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1999.

[30] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[31] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. 1975.

[32] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exper.*, 1993.