Bridging the Tenant-Provider Gap in Cloud Services

Virajith Jalaparti[†^] jalapar1@illinois.edu
Hitesh Ballani[†] hiballan@microsoft.com
Paolo Costa[†‡] costa@imperial.ac.uk
Thomas Karagiannis[†] thomkar@microsoft.com
Ant Rowstron[†] antr@microsoft.com

[†]Microsoft Research Cambridge, UK
[^]University of Illinois Urbana, IL, USA
[‡]Imperial College London, UK

ABSTRACT

The disconnect between the resource-centric interface exposed by today's cloud providers and tenant goals hurts both entities. Tenants are encumbered by having to translate their performance and cost goals into the corresponding resource requirements, while providers suffer revenue loss due to uninformed resource selection by tenants. Instead, we argue for a "job-centric" cloud whereby tenants only specify high-level goals regarding their jobs and applications. To illustrate our ideas, we present Bazaar, a cloud framework offering a job-centric interface for data analytics applications.

Bazaar allows tenants to express high-level goals and predicts the resources needed to achieve them. Since multiple resource combinations may achieve the same goal, Bazaar chooses the combination most suitable for the provider. Using large-scale simulations and deployment on a Hadoop cluster, we demonstrate that Bazaar enables a symbiotic tenant-provider relationship. Tenants achieve their performance goals. At the same time, holistic resource selection benefits providers in the form of increased goodput.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks]: Network Operations
General Terms: Algorithms, Design, Performance
Keywords: Cloud, Provider Interface, Resource Malleability, Job-centric, Resource selection, Cloud pricing

1. INTRODUCTION

The resource elasticity offered by today's cloud providers is often touted as a key driver for cloud adoption. Providers expose a minimal interface– users or *tenants* simply ask for the compute instances they require and are charged on a pay-as-you-go basis. Such resource elasticity enables elastic application performance; tenants can demand more or less compute instances to match performance needs.

While simple and elegant, there is a disconnect between this resource-centric interface exposed by providers and what tenants actually require. Tenants are primarily interested in predictable performance and costs for their applications [2, 28]; for instance, tenants want to satisfy constraints regarding their application completion time [9, 14, 20, 35, 39]. With today's setup, tenants bear the burden of translating these high-level goals into the corresponding resource requirements. Motivated by this disconnect, we argue for refactoring the tenant-provider relationship to improve cloud usability. Instead of today's resource-centric interface, providers should offer a *job-centric interface* whereby tenants only specify performance and cost goals for their jobs and applications.

Such a job-centric interface requires the provider to map tenant goals into resource requirements. While burdening the provider, a job-centric interface also opens up opportunities for them. Providers now have flexibility regarding *which* and *how many* resources to dedicate to a job and *when* to do so. As an example, many cloud applications are malleable in their resource requirements with multiple resource combinations yielding the same performance; for instance, to achieve a completion time goal, a MapReduce job can be run on a few virtual machines (VMs) with a lot of network bandwidth between them, a lot of VMs with a little network bandwidth, or somewhere in between these extremes. Similarly, a job to be completed in x hours could be finished much sooner using idle resources, thus allowing the provider to better serve subsequent tenants. The rigidity of today's resource-centric interface does not permit such provider flexibility.

In this paper, we take a first stab at a job-centric interface for cloud datacenters. Our examination of typical cloud applications from three different domains (data analytics, web-facing and HPC) shows that they exhibit *resource malleability*. This, along with *performance predictability*, is a key condition that our target applications must satisfy. For such applications, we devise mechanisms to automatically choose the resource combination, among those that can achieve tenant performance goals, that is most suitable for the provider. Thus, the impetus of this paper is on *satisfying tenant goals while capitalizing on the flexibility offered by a job-centric setup.*

We illustrate our mechanisms in the context of data analytics by focusing on MapReduce as an example cloud application. We design Bazaar, a cloud framework that takes tenant constraints regarding the completion time of their MapReduce job, and determines the resource combination most amenable to the provider that satisfies the constraints. Our choice of MapReduce was motivated by the fact that data analytics represent a significant workload for cloud infras-

SOCC'12, October 14-17, 2012, San Jose, CA USA

tructures [41,44], with some multi-tenant datacenters having entire clusters dedicated to running them [5,41]. Further, the malleability and predictability conditions mentioned above hold very well for MapReduce.

A key challenge in enabling a job-centric interface is shared datacenter resources. While tenants get dedicated compute instances with today's cloud offerings, other resources like the internal network and the cloud storage are shared and their performance can vary significantly [26,30,45]. This can lead to unpredictable application performance [15, 30, 36]. The problem of unpredictable performance has prompted efforts to provide guaranteed performance atop shared datacenter resources like the internal network [2, 13] and cloud storage [12, 21, 32]. These proposals allow tenants to explicitly request for resources beyond compute instances. Such multi-resource elasticity makes a job-centric interface feasible. Hence, we build upon these efforts to ensure tenant goals are met.

Though our core ideas apply to general multi-resource settings, we begin by focusing on two specific resources, compute instances (N) and the network bandwidth (B) between them. Bazaar uses a performance prediction component to determine the resource tuples $< N, B >$ that can achieve the completion time desired by the tenant. Since multiple resource tuples may achieve the same completion time, they are ranked in terms of the provider's cost to accommodate the tuple. Bazaar selects the resource tuple with the least cost to the provider, thus improving provider efficiency.

While a lot of recent work looks at performance prediction for analytics jobs [9, 10, 14, 29, 35, 37, 39, 46], the prediction problem for Bazaar is both simpler and harder. Avoiding shared resources by explicitly dedicating network bandwidth to jobs makes prediction simpler. However, exploring multiple resource combinations requires fast prediction. We adopt a hybrid approach (simple model + profiling) to predict how the performance of a MapReduce job scales in terms of multiple resources. Experiments show that the prediction is fast and has good accuracy (<12% average error). To counter outliers and failures that hinder perfect prediction, Bazaar relies on "slack" by finishing jobs earlier than desired. Such slack allows Bazaar to satisfy tenant goals in spite of imperfect prediction.

To demonstrate the feasibility of our approach, we built a Bazaar prototype and evaluate it on Emulab running Hadoop MapReduce jobs. To study the performance at scale, we complement these experiments with large-scale simulations. The results show that smart resource selection to satisfy tenant goals can yield significant gains for the provider. The provider can accept 3-14% more requests. Further, bigger (resource intensive) requests can be accepted which improves the datacenter goodput by 7-87%. Overall, Bazaar's higher-level tenant-provider interface benefits both entities. Tenants achieve their goals, while the resource selection flexibility improves datacenter goodput and hence, provider revenue.

In summary, this paper makes the following contributions:

- We propose a job-centric cloud interface that decouples the performance and cost goals of tenants from the underlying resource allocation. This better aligns the interests of tenants and the cloud provider.
- We measure the malleability of representative cloud applications, and show that different combinations of compute and network resources can achieve the same application performance.
- We devise a metric for the cost of accommodating multi-resource requests from the provider's perspective. This allows one resource tuple to be compared against another.
- We extend our metric to exploit malleability along the *time domain*, i.e., finishing jobs earlier than required by dedicating them idle resources. We find this can further reduce median job completion time by more than 50%.
- We present the design, implementation and evaluation of Bazaar, and illustrate how tenant performance goals can be met in a multi-resource setting.

While this paper intentionally focuses on completion time goals, a job-centric interface can also accommodate cost goals. In Section 6, we briefly discuss a novel job-centric pricing model that, when coupled with Bazaar, can allow tenants to achieve their cost goals too.

2. BACKGROUND AND MOTIVATION

Cloud providers today allow tenants to ask for virtual machines or VMs on demand. The VMs can vary in size; small, medium or large VMs are typically offered reflecting the available processing power, memory and local storage. For ease of exposition, the discussion here assumes a single VM class. A tenant request can thus be characterized by N, the number of VMs requested. Tenants pay a fixed amount per hour per VM; thus, renting N VMs for T hours costs $\$k_v * NT$, where k_v is the hourly VM price. For Amazon EC2, k_v = \$0.08 for small VMs.

Data analytics in the cloud. Analysis of big data sets underlies many web businesses [41, 43, 44] migrating to the cloud. Parallel frameworks like MapReduce [6], Scope [5], Dryad [16] cater to such data analytics, and form a key component of cloud workloads. Despite a few differences, these frameworks are broadly similar and operate as follows. Each job typically consists of three phases, (i). reading input data and applying a grouping function, (ii). shuffling intermediate data among the compute instances across the network, (iii). applying an aggregation function to generate the final output. For example, in the case of MapReduce, these phases are known as *map*, *shuffle*, and *reduce* phases. Computation may involve a series of such jobs.

Predictable performance. The parallelism provided by data parallel frameworks is an ideal match for the resource elasticity offered by cloud computing since the completion time of a job can be tuned by varying the resources devoted to it. Tenants often have high-level performance or cost requirements for their data-analytics. Such requirements may dictate, for example, that a job needs to finish by a certain time. However, with today's setup, tenants are responsible for mapping such high-level completion time goals down to specific resources needed for their jobs. While several recent efforts tackle this problem [9, 14, 35, 39], most of them focus only on the number and kind of VMs (or slots) required, and none account for the shared network.

Yet, the performance of most data analytic jobs depends on factors well-beyond the number of VMs devoted to them. For instance, apart from the actual processing of data, a job running in the cloud also involves reading input data off the cloud storage service and shuffling data between VMs over

the internal network. Since the storage service and the internal network are shared resources, their performance can vary significantly [26, 30]. This, in turn, impacts application performance. For instance, Schad et al. [30] found that the completion time of the same job executing on the same number of VMs on Amazon EC2 can vary considerably, with the underlying network contributing significantly to the variation. Thus, *without accounting for resources beyond simply the compute units, the goal of determining the number of VMs needed to achieve a desired completion time is intractable.*

Multi-resource elasticity. Performance issues with shared resources, such as the ones described above, prompted a slew of proposals that offer guaranteed performance atop such resources [2, 12, 13]. With these, tenants can request resources beyond just VMs; for instance, tenants can specify the network bandwidth between their VMs [2, 13]. We note that providing tenants with a guaranteed amount of individual resources makes the problem of achieving high-level performance goals tractable.

Our approach leverages such multi-resource elasticity and builds upon efforts that provide guaranteed resources. We consider a two-resource setting whereby the provider can dedicate VMs and a network slice to tenant jobs. As proposed in [2], a tenant's resources are characterized by a two tuple $<N, B>$ which gives the tenant N VMs, each with an aggregate network bandwidth of B Mbps to other VMs of the same tenant. However, before discussing how such resource elasticity can be exploited in a job-centric setup, we first quantify its impact on typical cloud applications.

2.1 Malleability of data-analytics applications

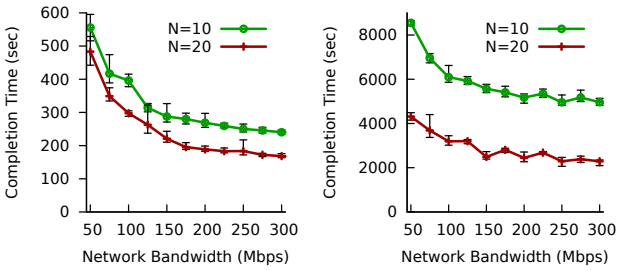
We first focus on data analytic frameworks, and use MapReduce as a running example. Our goal is to study how its performance is affected when varying different resources.

Hadoop Job	Input Data Set
`Sort`	200GB using Hadoop's RandomWriter
`WordCount`	68GB of Wikipedia articles
`Gridmix`	200GB using Hadoop's RandomTextWriter
`TF-IDF`	68GB of Wikipedia articles
`LinkGraph`	10GB of Wikipedia articles

Table 1: MapReduce jobs and the size of their input data.

We experimented with the small yet representative set of MapReduce jobs listed in Table 1. These jobs capture the use of data analytics in different domains and the varying complexity of such workloads (through multi-stage jobs). `Sort` and `WordCount` are popular for MapReduce performance benchmarking, not to mention their use in business data processing and text analysis respectively [38]. `Gridmix` is a synthetic benchmark modeling production workloads, Term Frequency-Inverse Document Frequency or `TF-IDF` is used in information retrieval, and `LinkGraph` is used to create large hyperlink graphs. Of these, `Gridmix`, `LinkGraph`, and `TF-IDF` are multi-stage jobs.

We used Hadoop MapReduce on Emulab to execute the jobs while varying the number of nodes devoted to them (**N**). We also used rate-limiting on the nodes to control the network bandwidth between them (**B**). For each $<N, B>$ tuple, we executed a job fives times to measure the completion time for the job and its individual phases. While the experiment setup is further detailed in §4.1, here we just focus on the performance trends.



(a) `LinkGraph` (b) `Sort`

Figure 1: Completion time for jobs with varying network bandwidth. Error bars represent Min–Max values.

Figure 1(a) shows the completion time for `LinkGraph` on a cluster with 10 and 20 nodes and varying network bandwidth. As the bandwidth between the nodes increases, the time to shuffle the intermediate data between map and reduce tasks shrinks, and thus, the completion time reduces. However, the total completion time stagnates beyond 250 Mbps. This is because the local disk on each node provides an aggregate bandwidth of 250 Mbps. Hence, increasing the network bandwidth beyond this value does not help since the job completion time is dictated by the disk performance. This is an artifact of the disks on the testbed nodes. If the disks were to offer higher bandwidth, increasing the network bandwidth beyond 250 would still shrink the completion time.

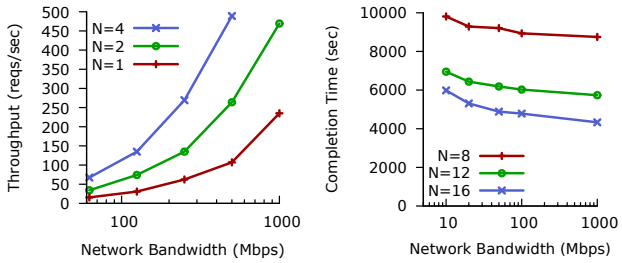
The same trend holds for the other jobs we tested. For instance, Figure 1(b) shows that the completion time for `Sort` reduces as the number of nodes and the network bandwidth between the nodes is increased. Note however that the precise impact of either resource is job-specific. For instance, we found that the relative drop in completion time with increasing network bandwidth is greater for `Sort` than for `WordCount`. This is because `Sort` is I/O intensive with a lot of data shuffled which means that its performance is heavily influenced by the network bandwidth between the nodes.

Apart from varying network bandwidth, we also executed the jobs with varying number of nodes. The results are detailed in §4.1 (Figures 6(a) and 6(b)) and show that the completion time for a job is inversely proportional to the number of nodes devoted to it. This is a direct consequence of the data-parallel nature of MapReduce.

2.2 Malleability of other cloud applications

The findings in the previous section extend to other cloud applications as well. We briefly discuss two examples here.

Three-tier, web application. We used a simple, unoptimized ASP.net web application with a SQL backend as a representative of web-facing workloads migrating to the cloud. We varied the number of nodes (N) running the middle application tier and the bandwidth (B) between the application tier (middle nodes) and the database-storage tier (backend nodes). We used the Apache benchmarking tool (`ab`) to generate web requests and determine the peak throughput for any given resource combination. Figure 2(a) shows that the application throughput improves in an expected fashion as either resource is increased.



(a) Web application (b) MPI application

Figure 2: Performance for cloud applications varies with both N and B.

MPI application. We used an MPI application generating the Crout-LU decomposition of an input matrix as an example of cloud HPC and scientific workloads. Figure 2(b) shows the completion time for a 8000x8000 matrix with varying N and B. Given the CPU-intensive nature of the application, increasing the number of nodes improves performance significantly. As a contrast, the impact of the network is limited. For instance, improving the bandwidth from 10 to 100 Mbps improves completion time only by 15-25%.

Overall, the experiments above lead to two expected yet key findings.
(1). The performance of typical cloud applications depends on resources beyond just the number of compute instances.
(2). They confirm the *resource malleability* of such applications; an application can achieve the same performance with different resource combinations, thus allowing one resource to be traded-off for the other. For instance, the throughput for the web application above with two nodes and 250 Mbps of network bandwidth is very similar to that with four nodes and 125 Mbps of network. Table 2 further emphasizes this for data analytic applications by showing examples where a number of different compute-node and bandwidth combinations achieve almost the same completion time for the `LinkGraph` and `WordCount` jobs. This flexibility is important in a job-centric cloud; it allows for improved cloud efficiency and hence, greater provider revenue.

Hadoop Job – Completion Time (sec)	<Nodes, Bandwidth (Mbps)> alternatives
`LinkGraph` – 300 (±5%)	<34, 75>, <20, 100> <10, 150>, <8, 250>
`LinkGraph` – 400 (± 5%)	<30, 60>, <10, 75> <8, 150>, <6, 200>
`WordCount` – 900 (± 3%)	<30, 45>, <20, 50> <10, 100>, <8, 300>
`WordCount` – 630 (± 3%)	<32, 50>, <20, 75> <14, 100>, <12, 300>

Table 2: Examples of `WordCount` and `LinkGraph` jobs achieving similar completion times with different resource combinations.

2.3 Scope and assumptions

In this paper, we aim to build a job-centric cloud interface. We identify two conditions that our target applications must satisfy:

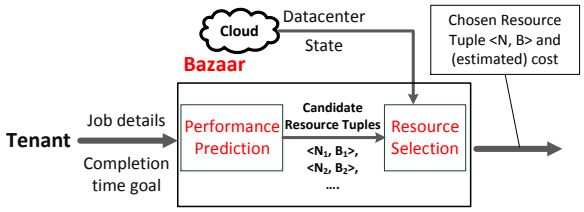


Figure 3: Bazaar offers a job-centric interface.

(1). *Performance predictability.* It should be feasible to predict the performance of the application when running on a given set of resources.
(2). *Resource malleability.* The application performance should vary with the resources dedicated to it. Further, it should be possible to trade-off one resource for the other without impacting performance.

To help ground our arguments, we hereon focus on MapReduce as an example cloud application. Based on this, we design Bazaar, a cloud framework that offers a job-centric interface to data analytics workloads. We note that MapReduce is a very suitable candidate for this exercise since– (i). it is popular as a data analytic framework and has significant presence in cloud workloads [41,44] and (ii). it satisfies both our conditions very well. Its performance scales with the two resources we consider (condition 2) and as we show later, the well-defined architecture of MapReduce lends itself well for automatic analysis and low-overhead profiling (condition 1). However, the core ideas in this paper can be extended both to applications beyond two resources, as shown in §4.4, and beyond MapReduce [27,33].

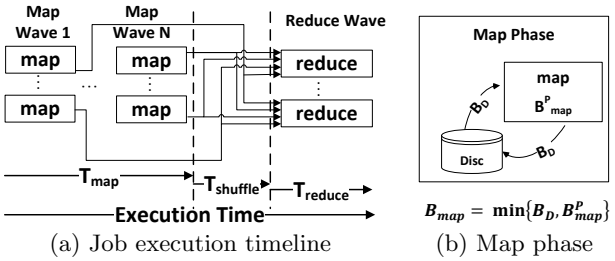
3. Bazaar DESIGN

Figure 3 shows Bazaar's design model. Tenants submit the specifications of their job, and high-level goals such as the job completion time and/or desired cost to Bazaar. In the context of MapReduce, the job specification includes the MapReduce program, input data size and a representative sample of the input data. Bazaar translates tenant goals to multiple *resource tuples* $<N, B>$, each comprising the number of VMs and the network bandwidth between the VMs. Since each of these resource tuples will finish the tenant's request on time, Bazaar's objective is to choose the one that allows the cloud provider to accept more requests in the future, thus maximizing its revenue. This selection is based on the current state of the datacenter. As shown in the figure, the translation of tenant goals into resource tuples uses two components–

(1). A *performance prediction* component that uses job details to predict the set of resource tuples that can achieve the desired completion time.
(2). A *resource selection* component that selects *which* resource tuple should be allocated. This choice is made so as to minimize the impact of the request on the provider's ability to accommodate future tenant requests.

3.1 Performance prediction

Translating tenant goals requires Bazaar to predict the completion time of a MapReduce job executing on a specific set of resources. Performance prediction has been extensively studied in a variety of domains such as operating systems [18], user software [4], and databases [22]. In the



(a) Job execution timeline (b) Map phase

Figure 4: Timeline of a MapReduce job and overview of the map phase.

context of MapReduce, efforts like Mumak [46] and MR-Perf [37] have built detailed MapReduce simulators that can be used for prediction. However, this results in non-trivial prediction times. To allow for an exploration of different resource combinations, Bazaar requires fast prediction. In designing Bazaar, we explicitly chose to simplify our prediction model, thereby favoring fast prediction over accuracy. This choice is also motivated by the observation that even the most accurate model will not be able to accurately account for all outliers. Instead, to compensate for any model inaccuracies, we introduce "slack" into our prediction. This helps us deal with common sources of prediction errors such as hardware heterogeneity and workload imbalance.

Inspired by profiling-based approaches for fast database query optimization [22], we capitalize on the well-defined nature of the MapReduce framework to predict the performance of MapReduce jobs. To this end, we design a prediction tool called *MRCute* or MapReduce Completion Time Estimator. MRCute adopts a hybrid approach for performance prediction by complementing an analytical model with job profiling. We first developed a high-level model of the operation of MapReduce jobs which yields a closed-form analytical expression for a job's completion time. This expression consists of job-specific and infrastructure-specific parameters. We determine these parameters by profiling the tenant job with a sample dataset on the provider's infrastructure.

At a high-level, the operation of MRCute can be summarized as follows. Given the program $\mathcal{P}$ for a MapReduce job, size of the input data $|I|$, a sample of the input data I_s, and a resource tuple $<N, B>$, MRCute estimates the job completion time:

$$MRCute(\mathcal{P}, |I|, I_s, N, B) \rightarrow T_{estimate}. \quad (1)$$

3.1.1 Job modeling

As shown in Figure 4(a), the execution of MapReduce jobs consists of three phases, each comprising multiple tasks. All tasks in a phase may not run simultaneously. Instead, the tasks execute in *waves*. For instance, the map phase in Figure 4(a) has N waves.

Typically, the three phases in a job execute sequentially. Hence, the completion time for a job is the sum of the time to complete individual phases, i.e., $T_{estimate} = T_{map} + T_{shuffle} + T_{reduce}$. The completion time for each phase depends on the number of waves in the phase, the amount of data consumed or generated by each task in the phase and the *phase bandwidth*. The phase bandwidth is the rate at which a given phase processes data. For instance, the completion time for the map phase is given by

$$T_{map} \quad = \quad W_{map} * \frac{I_{map}}{B_{map}},$$

where W_{map} is the number of waves in the phase, I_{map} is the data consumed by each map task and B_{map} is the map phase bandwidth. Of these, it is particularly challenging to determine the phase bandwidth. Since each phase uses multiple resources (CPU, disk, network), the slowest or the bottleneck resource governs the phase bandwidth. To determine the bandwidth for individual phases, and hence, the completion time of a job, we develop an analytical model by applying bottleneck analysis [24] to the MapReduce framework.

For example, during the **map phase** (Figure 4(b)), each map task reads its input off the local disk (assuming the input is locally available), applies the map function and writes the intermediate data to local disk. Thus, a map task involves the disk and CPU, and the map phase bandwidth is governed by the slowest of the two resources. Hence, $B_{map} = Min\{B_D, B^{\mathcal{P}}_{map}\}$, where B_D is the disk I/O bandwidth per map task and $B^{\mathcal{P}}_{map}$ is the rate at which data can be processed by the map function of the program $\mathcal{P}$ (assuming no other bottlenecks). Similarly, we derive expressions for the bandwidth for the shuffle and the reduce phase, and consequently, a closed-form expression for the job's completion time. To simplify exposition, the complete description of the analytical model is provided in the Appendix.

3.1.2 Job Profiling

Besides the input parameters specified in eq. 1, our closed-form expression for a job's completion time involves two other types of parameters: i). Parameters specific to the MapReduce configuration, such as the map slots per VM, which are known to the provider. ii). Parameters that depend on the infrastructure and the actual tenant job. These include the *data selectivity* of map and reduce tasks (S_{map} and S_{reduce}), the map and reduce phase bandwidths (B_{map} and B_{reduce}), and the physical disk bandwidth (B_D).

To determine the latter set of parameters, we profile the MapReduce program $\mathcal{P}$ by executing it on a single machine using a sample of the input data I_s. The profiler determines the execution time for each task and each phase, the amount of data consumed and generated by each task, etc. All this information is gathered from the log files generated during execution, and is used to determine the data selectivity and bandwidth for each phase. Concretely,

$$Profiler(\mathcal{P}, I_s) \rightarrow \{S_{map}, S_{reduce}, B_{map}, B_{reduce}, B_D\}.$$

For instance, the ratio of the data consumed by individual map tasks to the map task completion time yields the bandwidth for the job's map phase (B_{map}). The reduce phase bandwidth is determined similarly. Since the profiling involves only a single VM with no network transfers, the observed bandwidth for the shuffle phase is not useful for the model. Instead, we measure the disk I/O bandwidth (B_D) under MapReduce-like access patterns, and use it to determine the shuffle phase bandwidth.

3.1.3 Candidate resource tuples

Bazaar uses MRCute to determine the resource tuples that can achieve the completion time desired by the tenant. This involves two steps. First, the tenant job is profiled to determine infrastructure-specific and job-specific parameters.

These parameters are then plugged into the analytical expression to estimate the job's completion time when executed on a given resource tuple $<N, B>$. The latter operation is low overhead and is repeated to explore the entire space for the two resources. In practice, we envision the provider will offer a few classes (in the order of five-ten) of internal network bandwidth which reduces the search space significantly. For each possible bandwidth class, Bazaar determines the number of compute instances needed to satisfy the tenant goal. These $< N, B >$ combinations are the *candidate resource tuples* for the tenant request.

3.1.4 Dealing with prediction errors

As with any profiling-based approach, MRCute assumes that the behavior observed during profiling is representative of actual job operation. For example, the sample data used for profiling should be representative and of sufficient size.[1] Similarly, the machine used for profiling should offer the same performance as any other machine in the datacenter. While physical machines in a datacenter often have the same hardware configuration, their performance can vary, especially disk performance [23]. Overall, these assumptions do not always hold and can result in prediction errors.

Further, tasks of a job can fail or can lag behind (*outliers*), causing the actual job execution to deviate from "ideal" behavior and hence, prediction errors. Mantri's analysis of outliers in production settings identified three main root causes [1]– (i) network contention, (ii) bad machines, and (iii) workload imbalance. We address the first two causes and mask the last one as follows:

Network contention. Bazaar dedicates both VMs and a network slice to each job. The latter guarantees the network bandwidth for VMs. This avoids inter-job network contention, thus avoiding outliers due to poor network performance.

Bad machines. Even on our small testbed, we found significant variability in the completion time of the same task on different machines. We tracked this to variable disk performance which is observed in production datacenters too [1]. To account for such variability, MRCute maintains statistics regarding the disk bandwidth of individual machines. In practice, this can be obtained by profiling the machines periodically, for instance, when they are not allocated to tenants. Our evaluation shows that this improves the prediction accuracy significantly.

Workload imbalance. The amount of data processed by tasks belonging to the same phase can vary significantly which, in turn, leads to outliers. It is difficult to quantify such imbalance during prediction.

Broadly speaking, it is very hard to account for all possible causes that can result in inaccurate prediction. Hence, we use slack to mask such inaccuracies. A slack of 10% means we actually estimate the resources required to complete the job in 90% of the desired time, thus allowing 10% slack for misprediction. In §4.2.3, we study how the provider can set the slack to ensure tenant goals are met even in the presence of outliers.

[1]If sample is too small, external factors such as the OS page cache can influence the measurements and the observed bandwidth will be different from that seen by the actual job. We use MapReduce configuration parameters regarding the memory dedicated to each task to determine the minimum size of the sample data.

3.2 Resource selection

A job-centric interface offers flexibility to the provider. Since all the candidate tuples for a job achieve similar completion times, the provider can select which resource tuple to allocate. Bazaar takes advantage of this flexibility by selecting the resource tuple most amenable to the provider's ability to accommodate subsequent tenants, thus maximizing the provider revenue. This comprises the two following sub-problems.

The **feasibility problem** involves determining the set of candidate resource tuples that can actually be allocated in the datacenter, given its current utilization. For our two dimensional resource tuples, this requires ensuring that there are both enough unoccupied VM slots on physical machines and enough bandwidth on the network links connecting these machines. Oktopus [2] presents a greedy allocation algorithm for such tuples which ensures that if a feasible allocation exists, it is found. We use this algorithm to determine *feasible resource tuples*.

The **resource selection problem** requires selecting the feasible resource tuple that maximizes the provider's ability to accept future requests. However, in our setting, the resources required for a given tuple depend not just on the tuple itself, but also on the specific allocation. As an example, consider a tuple $<4, 200>$ requiring 4 VMs each with 200 Mbps of network bandwidth to other VMs. If all these VMs are allocated on a single physical machine, no bandwidth is required on the network link for the machine. On the other hand, if two of the VMs are allocated on one machine and two on another machine, the bandwidth required on the network links between them is 400 Mbps (2*200 Mbps).

To address this, we use the above allocation algorithm to convert each feasible resource tuple to a utilization vector capturing the utilization of physical resources in the datacenter after the tuple has been allocated. Specifically,

$$\text{Allocation}(< N, B >) \rightarrow U =< u_1, \ldots, u_d >,$$

where U is a vector with the utilization of all datacenter resources, i.e., all physical machines and links. The vector cardinality d is the total number of machines and links in the datacenter. For a machine k, u_k is the fraction of the VM slots on the machine that are occupied while for a link k, u_k is the fraction of the link's capacity that has been reserved for the allocated VMs.

Overall, given the set of utilization vectors corresponding to the feasible tuples, *the objective is to select the resource tuple that will minimize the number of rejected requests in the future.* This problem has been studied in various contexts, such as online ad allocation [8] and online routing and admission control in virtual circuit networks [19]. Depending on the context, one can show that different cost functions (that measure the cost for accepting a request) yield optimal scheduling for different request allocation models [3]. We experimented with a number of such cost functions and found that a cost function that captures the resource imbalance caused by the allocation of a resource tuple performs very well in terms of minimizing rejected requests. In our setting, *minimizing resource imbalance* translates to choosing the utilization vector that balances the capacity left across all resources after the request has been allocated. Precisely,

our selection heuristic aims to minimize the following

$$\text{minimize} \quad \sum_{j=1}^{d}(1-u_j)^2.$$

Hence, the resource imbalance is defined as the square of the fractional under-utilization for each resource. The lower this value, the better the residual capacity across resources is balanced. In literature, this is referred to as the Norm-based Greedy heuristic [25]. An extra complication in our setting is the hierarchical nature of typical datacenters. This leads to a hierarchical set of resources corresponding to datacenter hosts, racks and pods. Below we detail how this heuristic is extended to such a setting.

3.2.1 Resource imbalance heuristic

The resource imbalance heuristic applies trivially to a single machine scenario. Consider a single machine with a network link. Say the machine has N^{max} VM slots of which N^{left} are unallocated. Further, the outbound link of the machine has a capacity B^{max} of which B^{left} is unallocated. The utilization vector for this machine is

$$< u_1, u_2 >=< 1-\frac{N^{left}}{N^{max}}, 1-\frac{B^{left}}{B^{max}} > .$$

Thus, the resource imbalance for the machine is

$$\sum_{j=1}^{2}(1-u_j)^2 = \left\{\frac{N^{left}}{N^{max}}\right\}^2 + \left\{\frac{B^{left}}{B^{max}}\right\}^2 .$$

Since physical machines in a datacenter are arranged in racks which, in turn, are arranged in pods, there is a hierarchy of resources in the datacenter. To capture the resource imbalance at each level of the datacenter, we extend the set of datacenter resources to include racks and pods. Hence, the datacenter utilization is given by the vector $< u_1, \ldots, u_m >$, where m is the sum of physical machines, racks, pods and links in the datacenter. For a rack k, u_k is the fraction of VM slots in the rack that are occupied and the same for pods. Hence, for a resource tuple being considered, the overall resource imbalance is the sum of the imbalance at individual resources, represented by set C, whose utilization changes because of the tuple being accepted, i.e., $\sum_{j\in C}(1-u_j)^2$.

A lower resource imbalance indicates a better positioned provider. Hence, Bazaar chooses the utilization vector and the corresponding resource tuple that minimizes this imbalance. Since the allocation algorithm is fast (median allocation time is less than 1ms), we simply try to allocate all feasible tuples to determine the resulting utilization vector and the imbalance it causes.

3.2.2 Resource selection example

We now use a simple example to illustrate how Bazaar's imbalance-based resource selection works. Consider a rack of physical machines, each with 2 VM slots and a Gigabit link. Also, imagine a tenant request with two feasible tuples $<N, B>$ (B in Mbps): $<3, 500>$ and $<6, 200>$. Figure 5 shows allocations for these two resource tuples. Network links in the figure are annotated with the (unreserved) residual bandwidth on the link after the allocation. The figure also shows the imbalance values for the resulting datacenter states. The former tuple has a lower imbalance and is chosen by Bazaar.

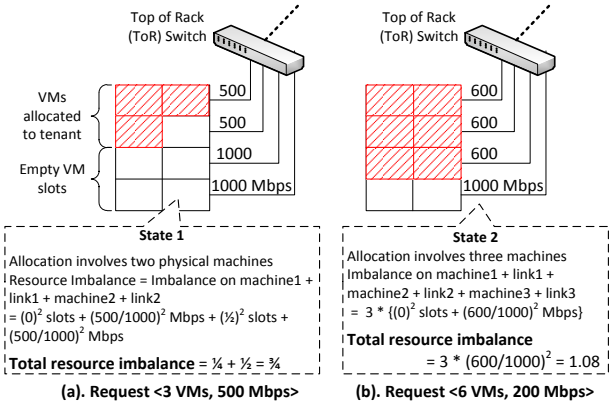


Figure 5: Selecting amongst two feasible resource tuples. Each physical machine has 2 VM slots and an outbound link of capacity 1000 Mbps. Each link is annotated with its residual bandwidth.

To understand this choice, we focus on the resources left in the datacenter after the allocations. After the allocation of the $<3, 500>$ tuple, the provider is left with five empty VM slots, each with an average network bandwidth of 500 Mbps (*state-1*). As a contrast, the allocation of $<6, 200>$ results in two empty VM slots, again with an average network bandwidth of 500 Mbps (*state-2*). We note that any subsequent tenant request that can be accommodated by the provider in state-2 can also be accommodated in state-1. However, the reverse is not true. For instance, a future tenant requiring the tuple $<3, 400>$ can be allocated in state-1 but not state-2. Hence, the first tuple is more desirable for the provider and is the one chosen by the resource imbalance metric.

4. EVALUATION

In this section, we evaluate Bazaar, focusing on its two main components, namely MRCute and resource selection. Our evaluation combines MapReduce experiments, simulations and a testbed deployment. Specifically:

(1). We quantify the prediction accuracy of MRCute. Results indicate that MRCute accurately determines the resources required to achieve tenant goals with low overhead and an average prediction error of less than 12% (§4.1).

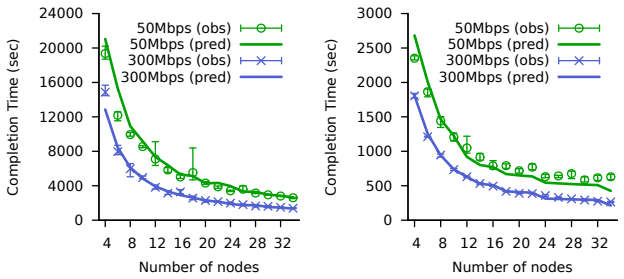
(2). We use large scale simulations to evaluate the benefits of Bazaar. Capitalizing on resource malleability significantly improves datacenter goodput (§4.2).

(3). We deploy and benchmark our prototype on a 26-node Hadoop cluster. We further use this deployment to cross-validate our simulation results (§4.3).

4.1 Performance prediction

We use MRCute to predict the job completion of the five MapReduce jobs described in §2.1 (Table 1). For each job, MRCute predicts the completion time for varying number of nodes (**N**) and the network bandwidth between them (**B**). The prediction involves profiling the job with sample data on a single node, and using the resulting job parameters to drive the analytical model.

To determine actual completion times, we executed each job on a 35-node Emulab cluster with Cloudera's distribution of Hadoop MapReduce (version 0.20.2). Each node has a quad-core Intel Xeon 2.4 GHz processor, 12 GB RAM and



(a) `Sort` (b) `WordCount`

Figure 6: Predicted completion time for `Sort` (an I/O intensive job) and `WordCount` (a CPU intensive job) matches the observed time.

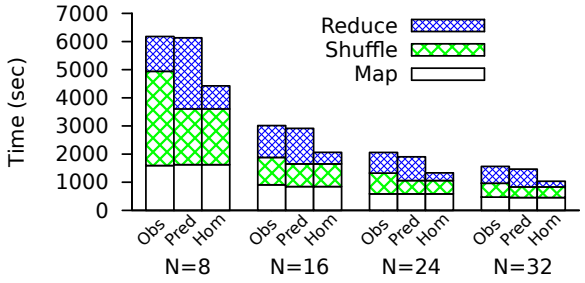


Figure 7: Per-phase breakdown of the observed (*Obs*) and predicted (*Pred*) completion time for `Sort` with bandwidth = 300 Mbps. *Hom* represents the predicted time assuming homogeneous disk performance.

a 1 Gbps network interface. The unoptimized jobs were run with default Hadoop configuration parameters. The number of mappers and reducers per node is 8 and 2 respectively, HDFS block size is 128 MB, and the total number of reducers is twice the number of nodes used. While parameter tuning can improve job performance significantly [14], our focus here is not improving individual jobs, but rather predicting the performance for a given configuration. Hence, the results presented here apply as long as the same parameters are used for job profiling and for the actual execution.

We first focus on the results for `Sort` and `WordCount`, two jobs at extreme ends of the spectrum. `Sort` is an I/O intensive job while `WordCount` is processor intensive. Figures 6(a) and 6(b) plot the observed and predicted completion time for five runs of these jobs when varying N and B. The figures show that the predicted and observed completion times are close throughout, with 8.9% prediction error on average for `Sort` and 20.5% at the 95^{th} percentile.

To understand the root cause of the prediction errors, we look at the per-phase completion time. Figure 7 presents this breakdown for `Sort` with varying number of nodes. The bars labeled *Obs* and *Pred* represent the observed and predicted completion time respectively. The figure shows that the predicted time for the map phase is very accurate; most of the prediction error results from the shuffle and reduce phases.

The reason for this difference in the prediction accuracy is that the map phase typically consists of a number of waves. Consequently, any *outlier* map tasks that are straggling in the earlier waves get masked by the latter waves and they do not influence the observed phase completion time significantly. In contrast, the shuffle and reduce phases execute in a single wave since the number of reduce tasks is the same as the number of reduce slots on the nodes. As a result, any outlier reduce tasks inflate the phase and in turn, the job completion time. Overall, such outliers introduce errors in the predicted completion time.

Beyond `Sort` and `WordCount`, the predicted estimates for the other jobs show similar trends. For brevity, we summarize the prediction errors in Figure 8. Overall, we find a maximum average error of 11.5% and a 95^{th} percentile of 20.5%.

4.1.1 *Accounting for outliers*

The basic MRCute model predicts job completion time assuming "ideal" operation. However, data analytics in production settings is far from ideal; nodes and tasks fail, and many tasks are outliers. While guaranteed network bandwidth avoids outliers due to network contention, here we describe how MRCute deals with outliers due to bad machines and quantify the impact of workload imbalance.

Bad machines. To account for disk performance variability, MRCute maintains statistics regarding the disk bandwidth of individual machines. To highlight the benefits of such benchmarking, the bars in figure 7 labeled *Hom* (Homogeneous) show the predicted times when MRCute does not account for disk performance heterogeneity, and instead, uses a constant value for the disk bandwidth in the analytical model. Since the performance of the disks on individual nodes varies, such an approach underestimates the reduce phase time which leads to a high prediction error.

Workload imbalance. The amount of data processed by tasks belonging to the same phase can vary significantly which, in turn, leads to outliers. To quantify the impact of such outliers on MRCute, we artificially introduced skew for the `Sort` job by choosing input keys from a skewed distribution. Here skew is the coefficient of variation ($\frac{stdev}{mean}$) for input across tasks belonging to the same phase. Figure 9 shows that the prediction error increases almost linearly with increasing skew. This is expected given that MRCute profiles the job only on sample data and does not explicitly account for input data skew. Mantri reported a median data skew of 0.34 which leads to 26% prediction error for MRCute. To account for such outliers resulting from workload imbalance and other factors, we use slack when determining resources for a job. As we show later, this allows us to satisfy tenant goals in the presence of outliers.

4.1.2 *Prediction overhead*

MRCute profiles a job on sample input data to determine the job parameters. This imposes two kinds of overhead.

(1). *Sample data.* We use information about the MapReduce configuration parameters, such as when data is spilled to the disk, to calculate the size of the sample data needed for the job. This is shown in Figure 8. Other than `Gridmix`, the jobs require <3 GB of sample data, a non-negligible yet small value compared to typical datasets used in data intensive workloads [7]. `Gridmix` is a multi-stage job with high selectivity. Hence, we need more sample data to ensure enough data for the last stage when profiling as data gets aggregated across stages. This overhead could be reduced by profiling individual stages separately but requires detailed knowledge about the input required by each stage.

(2). *Profiling time.* Figure 8 also shows the time to profile individual jobs. For `Sort` and `WordCount`, the profiling takes

Hadoop Job	Stages	Sample Data Size	Profiling Time	Prediction error (all runs)	
				Average	95%ile
`Sort`	1	1GB	100.8s	8.9%	20.5%
`WordCount`	1	450MB	67.5s	8.4%	19.7%
`Gridmix`	3	16GB	546s	11.5%	17.8%
`TF-IDF`	3	3GB	335s	5.6%	9.7%
`LinkGraph`	4	3GB	554.8s	8.2%	12.3%

Figure 8: Prediction overhead and error of MRCute for Hadoop jobs.

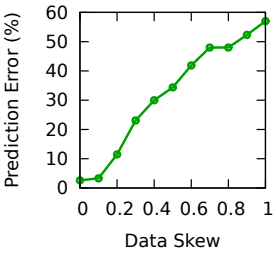


Figure 9: Impact of skew.

around 100 seconds. For the multi-stage jobs, profiling time is higher since more data needs to be processed. However, a job needs to be profiled only once to predict the completion time for all resource tuples, and we can also use information from past runs.

To summarize, these experiments indicate that MRCute can indeed generate good completion time estimates for MapReduce jobs.

4.2 Resource selection

Performance prediction allows Bazaar to determine the candidate resource tuples that can satisfy a tenant's completion time goals. Here, we evaluate the potential gains resulting from smart selection of the resource tuple to use.

4.2.1 Simulation setup

Given the small size of our testbed, we developed a simulator to evaluate Bazaar at scale. The simulator coarsely models a multi-tenant datacenter. It uses a three-level tree topology with no path diversity. Racks of 40 machines with one 1 Gbps link each and a Top-of-Rack switch are connected to an aggregation switch. The aggregation switches, in turn, are connected to the datacenter core switch. The results in the following sections involve a datacenter with 16,000 physical machines and 4 VMs per machine, resulting in a total of 64,000 VMs. The network has an oversubscription of 1:10 and we vary this later. Each VM has a local disk. While high-end SSDs can offer bandwidth in excess of 200 MB/s for even random access patterns [47], we conservatively use a disk I/O bandwidth of 125 MB/s = 1 Gbps such that it can saturate the network interface.

MapReduce jobs. We use a simple model for MapReduce jobs. The program $\mathcal{P}$ associated with a job is characterized by four parameters– the rate at which data can be processed by the map and reduce function when there are no I/O bottlenecks ($B^{\mathcal{P}}_{map}$, $B^{\mathcal{P}}_{reduce}$) and the selectivity of these functions (S_{map}, S_{reduce}). Given the input size, the selectivity parameters are used to determine the size of the intermediate and output data generated by the job. Note that an I/O intensive job like `Sort` can process data fast and has high values for $B^{\mathcal{P}}_{map}$ and $B^{\mathcal{P}}_{reduce}$. To capture the entire spectrum of MapReduce jobs, we choose these parameters from an exponential distribution with a mean of 500 Mbps. We also experiment with other mean values.

Tenant Requests. Each tenant request consists of a MapReduce job, input size and a completion time goal. This information is fed to the analytical model to determine the candidate resource tuples for the job. From these candidate tuples, one tuple $<N, B>$ is chosen based on the selection strategies described below. The corresponding resources, N VMs with B Mbps of network bandwidth, are allocated using the allocation algorithm in [2]. If the request cannot be allocated because of insufficient resources, it is rejected.

We simulate all three phases of MapReduce jobs. We do not model the disk and CPU operations. Instead, the duration of the map and the reduce phase is simply calculated a priori by invoking the MRCute analytical model. As part of the shuffle phase, we simulate all-to-all traffic matrix with N^2 network flows between the N VMs allocated to the tenant. Given the bandwidth between VMs, we use max-min fairness to calculate the rate achieved by each flow. The shuffle phase completes when all flows complete.

Resource selection strategies. We evaluate three strategies to select a resource tuple.

(1). *Baseline.* This strategy does not take advantage of a job's resource malleability. Instead, one of the candidate tuples is designated as the baseline tuple $<N_{base}, B_{base}>$. The job is executed using this baseline resource tuple.

(2). *Bazaar-R* (random selection). A tuple is randomly selected from the list of candidates, and if it can be allocated in the datacenter, it is chosen. Otherwise the process is repeated. This strategy takes advantage of resource malleability to accommodate requests that otherwise would have been rejected. However, it does not account for the impact that a tuple bears on the provider.

(3). *Bazaar-I* (imbalance-based selection). For each tuple, we determine how it would be allocated and calculate the resulting utilization vector and resource imbalance. The tuple with the lowest resource imbalance is chosen.

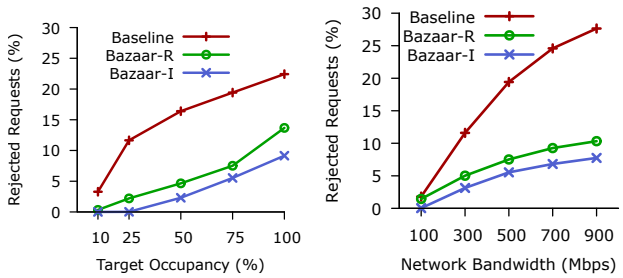
Workload. To model the operation of cloud datacenters, we simulate tenant requests arriving over time. By varying the tenant arrival rate, we vary the *target VM occupancy* for the datacenter. Assuming Poisson tenant arrivals with a mean arrival rate of λ, the target occupancy on a datacenter with M total VMs is $\frac{\lambda N T}{M}$, where T is the mean completion time for the requests and N is the mean number of requested VMs in the Baseline scenario.

4.2.2 Selection benefits

We simulate the arrival and execution of 15,000 tenant requests. The desired completion time for each request is chosen such that the number of compute nodes (N_{base}) and network bandwidth (B_{base}) required in the Baseline scenario is exponentially distributed. The mean value for N_{base} is 50, which is consistent with the mean number of VMs that tenants request in cloud datacenters [31].

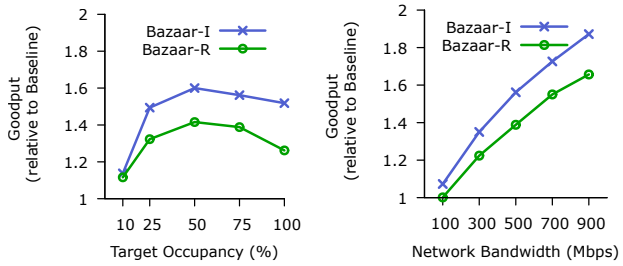
Workloads and metrics. Two primary variables are used in the following experiments to capture different workloads. First, we vary the *mean bandwidth* required by tenants (B_{base}). This reflects tenants having varying completion time requirements. Second, we vary the *target occupancy* to control the tenant request arrival rate.

From a provider's perspective, we look at two metrics to quantify the potential benefits of resource selection. First



(a) Mean BW = 500Mbps (b) Target Occupancy = 75%

Figure 10: Percentage of rejected requests, varying mean bandwidth and target occupancy.



(a) Mean BW = 500Mbps (b) Target Occupancy = 75%

Figure 11: Datacenter goodput with varying mean bandwidth and varying target occupancy.

is the fraction of requests that are rejected. However, this, by itself, does not represent the full picture since individual requests are of different sizes, i.e., each request processes a different amount of data. To capture this, we also look at the sum of input data consumed across all requests. This represents the total useful work in the datacenter and is, thus, termed as the *datacenter goodput*.

Impact of varying mean bandwidth and target occupancy. Figure 10(a) plots the percentage of rejected requests with varying target occupancy. For all selection strategies, the rejection ratio increases with increasing target occupancy. This is because requests start arriving faster and, hence, a greater fraction have to be rejected. The figure shows that, depending on the occupancy, Bazaar-I results in 3-14% fewer requests being rejected. Bazaar-R rejects around 2-5% more requests than Bazaar-I. However, as we explain below, the actual benefit of the imbalance-based selection is larger.

To put this in perspective, operators like Amazon EC2 target an average occupancy of 70-80% [42]. Figure 10(b) plots the rejected requests for a target occupancy of 75%. The figure shows that the difference between the fraction of requests rejected by both Bazaar strategies as compared to Baseline increases with increasing mean bandwidth. Increasing the bandwidth required by the job implies tighter completion time requirements which, in turn, means there are greater gains to be had from selecting the appropriate resource combination. At mean bandwidth of 900 Mbps, Bazaar-I rejects 19.9% fewer requests than Baseline.

Figure 11 shows the datacenter goodput for the Bazaar selection strategies relative to Baseline. Depending on the occupancy and bandwidth, *Bazaar-I improves the goodput by 7-87% over Baseline, while Bazaar-R provides improvements of 0-66%.* As an example, at typical occupancy of 75% and a mean bandwidth of 500 Mbps, Bazaar-I and Bazaar-R offer 56% and 39% benefits relative to Baseline respectively. Note that the gains with Bazaar-R show how resource malleability can be used to accommodate tenant requests that would otherwise have been rejected. The further gains with Bazaar-I represent the benefits to be had by smartly selecting the resources to use.

In figure 11(a), the relative improvement in goodput with Bazaar strategies first increases with target occupancy and then declines. This is because, at both low and high occupancy, there is not as much room for improvement. At low occupancy, requests arrive far apart in time and most can also be accepted by Baseline. At high occupancy, the arrival rate is high and the datacenter is heavily utilized. In figure 11(b), the gains increase with increasing bandwidth. As explained above, this results from shrinking completion time requirements which allow Bazaar strategies to accept more requests as compared to Baseline. Further, Bazaar is able to accept bigger requests resulting in even higher relative gains.

Impact of simulation parameters. We also determined the impact of other simulation parameters on Bazaar performance and the results stay qualitatively the same. Due to space constraints, we only show the results of varying oversubscription and briefly discuss the impact of varying the mean disk and other parameters.

Figure 12 shows the relative goodput with varying network oversubscription. *Even in a network with no oversubscription*, e.g., [11], *Bazaar-I is able to accept 10% more requests and improves the goodput by 27% relative to Baseline.* Further, the relative improvement with Bazaar increases with increasing oversubscription before flattening out. This is because the physical network becomes more constrained and Bazaar can benefit by reducing the network requirements of tenants while increasing their VMs.

We also ran experiments using different values of the disk bandwidth. As expected, low values of the disk bandwidth reduce the benefits of Bazaar-I. When the disk bandwidth is extremely low (250 Mbps), increasing the network bandwidth beyond this value does not improve performance. Thus, there are very few candidate resource tuples and the gains with Bazaar are small (2% over Baseline). However, as the disk bandwidth improves, there are more candidate tuples to choose from and the performance improves.

Finally, we varied the mean task bandwidth (map and reduce) and also the datacenter size (up to a maximum of 32,000 servers and 128,000 VMs) and the results confirmed the trends observed in Figure 10 and 11.

Comparison with today's setup. Today, cloud providers do not offer any bandwidth guarantees to tenants. VMs are allocated using a greedy locality-aware strategy and bandwidth is fair-shared across tenants using TCP. In Figure 13(a), we compare the performance of Bazaar-I against a setup representative of today's scenario, which we denote as *Fair-sharing*. For low values of occupancy, Fair-sharing achieves a slight better performance than Bazaar-I. The reason is that Bazaar-I reserves the network bandwidth throughout the entire duration of a tenant's job. This also includes the map and reduce phase, which are typically characterized by little or no network activity. In contrast, in Fair-sharing, the network bandwidth is not exclusively assigned to tenants and, hence, due to greater multiplexing, it

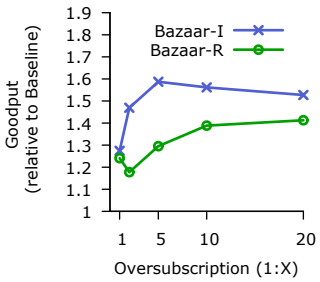
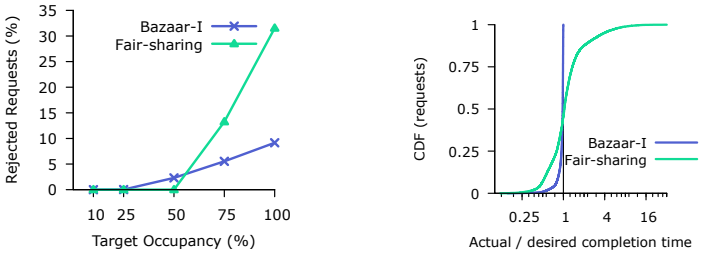
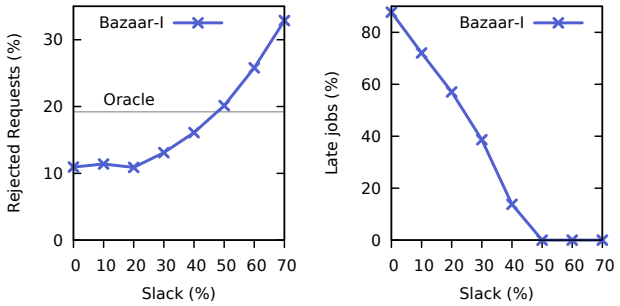


Figure 12: Network oversubscription (occupancy is 75%).



(a) Rejected requests. (b) CDF of completion time.

Figure 13: Comparing against today's setup (Mean BW is 500 Mbps).



(a) Rejected requests (b) Late jobs

Figure 14: Using *slack* to offset impact of outliers.

achieves a higher network utilization. Yet, for high values of occupancy, which are typical of today's datacenters [42], rejected requests significantly increase. This is due to the high congestion incurred in the core of the network, caused by the sub-optimal placement of VMs and corresponding flows.

The main drawback of Fair-sharing, however, is highlighted in Figure 13(b), which shows that the completion time is extended for at least 50% of the jobs and for 12% of the jobs the actual completion time is at least twice the desired completion time. Since tenants pay based on the time they occupy VMs, this also inflates tenant costs.

4.2.3 *Mitigating outliers with slack*

The results above assume perfect prediction. Since our prediction does not account for all possible outliers, the predicted completion time for a job can be off which, in turn, would cause the job to be *late*. To counter this, Bazaar relies on slack. To evaluate how much slack is needed in practice, we use the outlier distribution from Bing's production clusters (reported in [1]) to introduce outlier tasks in our experiments. Such tasks extend jobs past the predicted completion time. Given this, we measure the impact of varying slack on rejected requests (relevant for the provider) and late jobs (relevant for the tenants).

Figure 14(a) shows that more requests are rejected with increasing slack. As slack increases, it is harder to accommodate requests since they need to be finished sooner and thus, require more resources. For slack less than 50%, Bazaar-I rejects fewer requests than an "Oracle" that can do perfect prediction but does not capitalize on resource selection. In effect, smart resource selection allows us to offset prediction inaccuracies. The same trends hold for goodput too.

Figure 14(b) shows the percentage of jobs that finish beyond the goal completion time. With no slack, ∼90% of jobs are late, a consequence of 90% of jobs having at least one outlier. As slack increases, the late requests decrease almost linearly, and with a slack of 50%, no requests are late. Overall, the slack parameter gives the provider a knob to satisfy tenant goals even in the presence of outliers at the expense of greater rejections. For instance, the provider may use historical job information to determine the amount of slack to provision so as to bound the probability of breached SLAs.

4.3 Deployment

We complement our simulation analysis with experiments on a small-scale Hadoop cluster using a prototype implementation of Bazaar. We deployed Bazaar on 26 Emulab servers, using the same hardware setup described in Section 4.1 and the Cloudera distribution of Hadoop. We used the Linux Traffic Control API on individual servers to enforce the rate limits.

We configured one of the testbed servers as the cluster head node and the rest of the servers as compute nodes. The head node is responsible of generating tenant requests and allocate them on the compute nodes. The workload consisted of 100 `Sort` job requests with an exponentially generated input data size (the mean value was 5.7 GB). Like in the previous experiments, we used a target occupancy of 75%, mean B_{base} of 500 Mbps and mean N_{base} equal to 9.

The goal of these experiments is threefold. First, we quantify the benefits of Bazaar. Second, we cross-validate the accuracy of our simulator. Finally, we verify the scalability of our implementation to allocate requests in a much bigger network.

Benefits. Figure 15 shows that Bazaar-I is able to accept 11.43% more `Sort` jobs than Baseline (i.e., 8 extra jobs) and increases goodput by 15.47%. This is despite limited opportunities– the deployment is small and the disk bandwidth available is low. Also, with Bazaar-I, less than 2% of the accepted requests completed later than expected. Note, however, that in these experiments we did not add any slack, which would have enabled all requests to complete on time.

Cross-validation. To validate the accuracy of our simulator, we replicated the same workload in the simulator, i.e. the same stream of jobs arrive in the simulator as on the testbed. Across all cases, the maximum difference in the number of accepted requests and goodput is approximately 5.12%, and 8.43% respectively. This cross-validation gives us confidence in our simulation results.

Scalability analysis. To evaluate the performance of our prototype at scale, we measured the time to allocate tenant requests on a datacenter with 128,000 VMs. This includes both the time to generate the set of candidate resource tuples using the analytical model (Section 3.1.1) and to select the resources (Section 3.2). This does not include the job

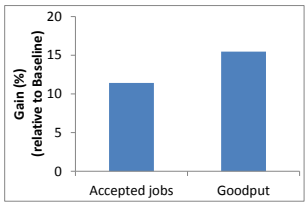


Figure 15: On testbed, Bazaar-I can accept more Hadoop jobs and increase goodput relative to Baseline.

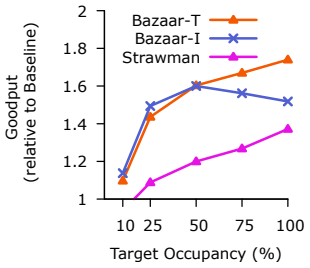


Figure 16: Datacenter goodput when exploiting time malleability.

profiling time. Over 10,000 requests, the median allocation time is 950.17 ms with a 99th percentile of 983.29 ms. Note that this only needs to be run when a tenant is admitted, and, hence, the overhead introduced is negligible.

4.4 Beyond two resources: time malleability

Our evaluation considered application malleability along two dimensions, i.e., N and B. However, a job-centric cloud allows the provider to exploit malleability across other resources and across time. Here, we briefly explore this opportunity. The provider can devote additional (idle) resources to tenant jobs, so that they complete before the desired time. In this way, the resources used by the job can be reclaimed earlier, and a larger number of requests can potentially be accommodated in the future. Tenants would benefit too since they would experience shorter than desired completion times.

We denote this further selection strategy as *Bazaar-T*. The key difference between Bazaar-T and Bazaar-I is that the latter only considers tuples $< N, B >$ that yield a completion time $T = T_{desired}$ while Bazaar-T also consider tuples where $T < T_{desired}$. Among these, Bazaar-T selects the tuple that minimizes the product of the tuple resource imbalance and T. Figure 16 shows that, at high values of the target occupancy, exploiting time flexibility significantly improves the ability of the provider to accommodate more requests and, hence, the goodput increases. Bazaar-T is also beneficial for tenants as *the median completion time is reduced by more than 50%, and for 20% of the jobs the completion time is reduced by 80%.* In Figure 16 we also consider a naive approach, *Strawman,* that always selects the tuple that yields the lowest completion time, irrespective of the resource imbalance. Such a strategy performs poorly as it tends to overprovision the resources for the early requests, which reduces the ability to accommodate future ones.

5. RELATED WORK

While we discussed a lot of relevant work in Section 2, we briefly comment on other efforts here. The prominence of data analytics has prompted a slew of proposals for improving analytics frameworks; for example, determining good configuration parameters [14, 20], better scheduling [17, 40], etc.

The observation that today's provider interface is disconnected from tenant needs has also led to recent efforts to tackle the problem [9, 14, 35, 39]. Elasticiser [14] and Conductor [39] translate tenant goals into cloud resource requirements, while Aria [35] and Jockey [9] focus on private settings. All these proposals involve a performance prediction component; Conductor uses a model while Jockey uses a simulator. Like Bazaar, Aria and Elasticiser combine profiling with modeling. Aria uses historical information for profiling and has a model that estimates performance bounds. Elasticiser profiles using instrumentation and relies on a statistical model.

The key difference in Bazaar is our focus on enabling a job-centric interface. Further, we account for multiple resources. Specifically, we dedicate a network slice to tenant jobs. The fact that the network is a shared yet distributed resource makes this hard. However, as explained below, doing so makes the translation of tenant goals simpler and benefits the provider. First, guaranteed network resources avoid inter-job contention and make model-based prediction tractable. Second, a per-tenant network slice, combined with the notion of slack, avoids the need for dynamic adaptation [9, 39]. Finally, by exploiting the trade-off between resources (VMs, network) and between time, the provider achieves greater flexibility and revenue.

There has also been a lot of progress towards data analytics performance prediction. For example, Mumak [46] and MRPerf [37] are discrete-event simulators for MapReduce, Ganapathi et al. [10] use statistical analysis to discover feature vectors and predict job performance, ParaTimer [29] is a job progress estimator that relies on debug runs of the job. Bazaar needs fast prediction, so it cannot use detailed simulators like MRPerf that take minutes per simulation [14]; exploring 100 resource tuples would take 100s of minutes. Instead, we adopt a hybrid approach that trades off accuracy for prediction speed. Tian et al. [34] use a similar tact involving profiling on sample data and then linear regression for prediction. Finally, there has been work towards performance prediction for other cloud applications (eg., web [27] and ERP applications [33]). Coupling our resource selection strategies with future prediction frameworks would allow Bazaar to offer a job-centric interface to applications beyond MapReduce.

6. PRICING DISCUSSION

Bazaar's job-centric interface achieves a better alignment of provider-tenant interests. This opens up interesting opportunities to investigate new pricing models.

Today, tenants pay based on the amount of time they use compute resources. For instance, with today's setup, the cost for a tenant renting N VMs for T hours is $\$k_v \cdot NT$, where k_v is the hourly VM price. Such *resource-based pricing* can be naively extended to multiple resources. A simple extension to a two-resource setting would be to charge for the network bandwidth too. So a tenant with N VMs and B Mbps of network bandwidth would pay $\$k_c \cdot NBT$, where k_c is the hourly combined price for VMs and bandwidth. However, this results in a mismatch of tenant and provider interests. The cheapest resource tuple to achieve the tenant's goal may not concur with the provider's preferred resource combination. Further, resource based pricing entails tenants have to pay based on a job's *actual* completion time. Hence, from a pricing perspective, there is a disincentive for the provider to reduce the completion time.

By decoupling the tenants from the underlying resources, Bazaar offers the opportunity of moving away from resource based pricing. Instead, tenants could be charged based only on the characteristics of their job, the input data size and the desired completion time. Such *job-based pricing* can ben-

efit both entities. Tenants specify *what* they desire and are charged accordingly; providers decide *how* to efficiently accommodate the tenant request based on job characteristics and current datacenter utilization. Further, since the final price does not depend on the completion time, providers now have an incentive to complete tenant jobs on time, possibly even *earlier* than the desired time as in Bazaar-T.

A job-centric cloud, coupled with job-based pricing, can thus enable a symbiotic tenant provider relationship where tenants benefit due to fixed costs upfront and better-than-desired performance while providers use the increased flexibility to improve goodput and, consequently, total revenue.

7. CONCLUSIONS

Bazaar enables a job-centric cloud interface for data analytics. It translates high-level tenant goals to datacenter resources. This involves predicting application resource requirements. Further, Bazaar exploits the resource malleability of cloud applications to select the resource combination that will achieve tenant goals while improving provider efficiency. Our evaluation shows that, by serving as a conduit for exchange of information between tenants and providers, Bazaar provides benefits for both entities.

Acknowledgements

We are grateful to Christos Gkantsidis, Greg O'Shea, and Bozidar Radunovic for their helpful comments and suggestions.

8. REFERENCES

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[3] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 2005.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, 2004.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Comm.of ACM*, 51(1),2008.

[8] N. R. Devanur, K. Jain, B. Sivan, and C. A. Wilkens. Near optimal online algorithms and fast approximation algorithms for resource allocation problems. In *EC*, 2011.

[9] A. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[10] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson. Statistics-Driven Workload Modeling for the Cloud. In *SMDB*, 2010.

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[12] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[13] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[14] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *ACM SOCC*, 2011.

[15] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. Technical report, Delft University of Technology, 2010.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[18] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *OSDI*, 2006.

[19] A. Kamath, O. Palmon, and S. Plotkin. Routing and admission control in general topology networks with poisson arrivals. In *ACM-SIAM SODA*, 1996.

[20] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.

[21] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Trans. Storage*, 1, 2005.

[22] M. Kremer and J. Gryz. A Survey of Query Optimization in Parallel Databases. Technical report, York University, 1999.

[23] E. Krevat, J. Tucek, and G. R. Ganger. Disks Are Like Snowflakes: No Two Are Alike. In *HotOS*, 2011.

[24] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queuing network models*. 1984.

[25] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating Heuristics for Virtual Machines Consolidation. Technical Report MSR-TR-2011-9, MSR, 2011.

[26] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *IMC*, 2010.

[27] A. Li, X. Zong, S. Kandula, X. Yand, and M. Zhang. CloudProphet: Towards Application Performance Prediction in Cloud. In *SIGCOMM (Poster)*, 2011.

[28] Michael Armburst et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UCB, 2009.

[29] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[30] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *VLDB*, 2010.

[31] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the Datacenter Network. In *NSDI*, 2011.

[32] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI*, 2012.

[33] D. Tertilt and H. Krcmar. Generic Performance Prediction for ERP and SOA Applications. In *ECIS*, 2011.

[34] F. Tian and K. Chen. Towards Optimal Resource Provisioning for Running MapReduce programs in Public Cloud. In *CLOUD*, 2011.

[35] A. Verma, L. Cherkasova, and R. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *ICAC*, 2011.

[36] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *Login*, 2008.

[37] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*, 2009.

[38] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[39] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *NSDI*, 2012.

[40] 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 *EuroSys*, 2010.

[41] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[42] Amazon's EC2 Generating 220M+ Annually. `http://bit.ly/8rZdu`.

[43] Big Data @ Foursquare . `http://goo.gl/FAmpz`.

[44] Hadoop Wiki: PoweredBy. `http://goo.gl/Bbfu`.

[45] Measuring EC2 system performance. `http://bit.ly/48Wui`.

[46] Mumak: Map-Reduce Simulator. `http://bit.ly/MoOax`.

[47] Tom's Hardware Blog. `http://bit.ly/rkjJwX`.

APPENDIX

As described in Section 3.1.1, the completion time for each phase of a map reduce job depends on the number of waves in the phase, the amount of data consumed or generated by each task in the phase and the phase bandwidth. Here we analytically determine these values, and use them to derive an expression for the job completion time.

1). Phase bandwidth. We described the modeling of the map phase in §3.1.1. Following the same logic for the **reduce phase**, $B_{reduce} = Min\{B_D, B^{\mathcal{P}}_{reduce}\}$. During the **shuffle phase** (Figure 17), reduce tasks complete two operations. Each reduce task first reads its partition of the intermediate

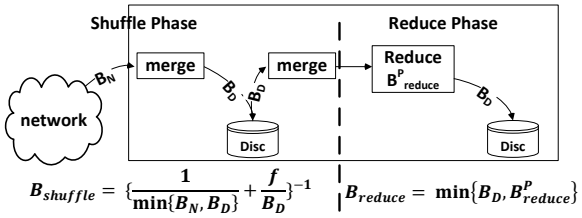


Figure 17: Detailed description of the resources involved in the shuffle and reduce phase.

data across the network, and then merges and writes it to disk. Hence, bandwidth $= Min\{B_D, B_N\}$, where B_N is the network bandwidth. Next, the data is read off the disk and an external merge sort is performed before the data is consumed by the reduce phase. This operation is bottlenecked at the disk, i.e., bandwidth $= B_D$. Note that the external merge sort can proceed in multiple steps leading to a larger amount of data being written to disk than that read from the map phase. We account for this through a factor f that captures the extra data written to disk relative to the data read in. Thus, we model the shuffle phase as two steps with bandwidths $Min\{B_D, B_N\}$ and B_D respectively.

2). Data consumed. For a MapReduce job with M map tasks, R reduce tasks and input of size $|I|$, each map task consumes $\frac{|I|}{M}$ bytes, while each reduce task consumes $\frac{|I|}{S_{map}*R}$ bytes and generates $\frac{|I|}{S_{map}*S_{reduce}*R}$ bytes with S_{map} and S_{reduce} being the *data selectivity* of map and reduce tasks respectively.

3). Waves. For a job using N VMs with M_c map slots per-VM, the maximum number of simultaneous mappers is $N * M_c$. Consequently, the map tasks execute in $\lceil \frac{M}{N*M_c} \rceil$ waves. Similarly, the reduce tasks execute in $\lceil \frac{R}{N*R_c} \rceil$ waves, where R_c is the number of reduce slots per-VM.

Since tasks belonging to a phase execute in waves, the completion time for a phase depends on the number of waves and the completion time for the tasks within each wave. Hence, for the map phase,

$$T_{map} = Waves_{map} * \frac{Input_{map}}{B_{map}} = \lceil \frac{M}{N*M_c} \rceil * \left\{ \frac{|I|/M}{B_{map}} \right\}.$$

Using similar logic for the shuffle and reduce phase completion time, the estimated job completion time is

$$\begin{aligned} T_{estimate} &= T_{map} + T_{shuffle} + T_{reduce} \\ &= \lceil \frac{M}{N*M_c} \rceil * \left\{ \frac{|I|/M}{B_{map}} \right\} \\ &+ \left\lceil \frac{R}{N*R_c} \right\rceil * \frac{|I|}{\{S_{map}*R\}} * \left\{ \frac{1}{\mathrm{Min}(B_D, B_N)} + \frac{f}{B_D} \right\} \\ &+ \left\lceil \frac{R}{N*R_c} \right\rceil * \left\{ \frac{|I|/\{S_{map}*S_{reduce}*R\}}{B_{reduce}} \right\}. \end{aligned}$$

The discussion above assumes that the map tasks are scheduled so that their input is available locally and the output generated by reducers is written locally with no further replication. Further, the reduce tasks are separated from the map phase by a barrier and execute once all the map tasks are finished [1]. While these assumptions helped the presentation, MRCute does not rely on them. For instance, to account for non data-local maps, the network bandwidth is also considered when estimating B_{map}.