Bazaar: Enabling Predictable Performance in Datacenters

Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, Antony Rowstron
MSR Cambridge, UK

Technical Report
MSR-TR-2012-38

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1. INTRODUCTION

The resource elasticity offered by 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 low-level interface exposed by providers and what tenants actually require. Tenants are primarily interested in predictable performance and costs for their applications [1,2]; for instance, satisfying constraints regarding their application completion time [3,4,5]. With today's setup, tenants bear the burden of translating these high-level goals into the corresponding resource requirements. Given the *multiplicity of resources* in a datacenter (compute instances, network, storage), such a mapping requires determining how application performance scales with individual resources, which is often non-trivial.

The difficulty of mapping tenant goals is further exacerbated by *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 [6,7,8]. This leads to unpredictable performance for a wide-variety of applications, including user-facing online services [9,7], data analytics [7,10] and HPC applications [11]. Therefore, the task of determining the resources needed to achieve tenant goals with today's setup is intractable.

Apart from hurting *cloud usability*, the disconnect between tenants and providers impedes *cloud efficiency* too. A lot of applications running in cloud datacenters are malleable in their resource requirements with multiple resource combinations yielding the same performance. For instance, a completion time goal for a MapReduce job may be achieved through 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. Tenants making resource choices in isolation so as to satisfy their goals can result in sub-optimal choices that reduce system throughput.

Overall, the skewed division of functionality between tenants and providers imposed by today's setup hurts both entities; tenants cannot achieve predictable performance while providers lose revenue due to inefficient operation.

The problem of unpredictable performance has prompted efforts to provide guaranteed performance atop shared datacenter resources like the internal network [2,12,13] and cloud storage [14,15]. These proposals allow tenants to explicitly request for resources beyond compute instances, thus enabling true multi-resource elasticity. While necessary, these techniques are not sufficient. Even with guaranteed resources, automatically inferring how the performance of an arbitrary application scales with various resources is hard. Expecting a programmer to explicitly understand and describe how the program scales with each resource is also impractical.

In this paper, we take a first stab at enabling predictable application performance in cloud datacenters. Our examination of typical cloud applications from three different domains (data analytics, web-facing, MPI) shows that they exhibit both *resource malleability* and *performance predictability*, two key conditions that our target applications must satisfy. For such applications, we devise mechanisms to choose the resource combination that can achieve tenant performance goals and is most suitable for the provider. Thus, the impetus of this paper is on *ensuring predictable performance while capitalizing on resource malleability.*

We illustrate these mechanisms in the context of data analytics by focusing on MapReduce as an example cloud application. We designed Bazaar, a system that takes tenant constraints regarding the completion time (or cost) for 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 infrastructures [16,17], with some multi-tenant datacenters having entire clusters dedicated to running them [18,19,20]. Further, the malleability and predictability conditions described above hold very well for MapReduce.

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 desired completion time. As 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 provider cost, thus improving provider efficiency. Overall, this paper makes the following contributions:

- 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 present a gray-box approach to predict how the performance of a MapReduce job scales in terms of multiple resources. The prediction is fast, low overhead and has good accuracy (<12% average error).

- We devise a metric for the cost of multi-resource requests from the provider's perspective. This allows one resource tuple to be compared against another.
- We present the design and implementation of Bazaar, and use it to illustrate how tenants can achieve their performance goals in a multi-resource setting.

Using extensive large-scale simulations and deployment on a small testbed, we 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%.

Since there are no well established pricing models for multi-resource requests like the ones considered here, this paper intentionally focuses on completion time goals and datacenter goodput. Towards the end of the paper, we briefly discuss a novel pricing model that, when coupled with Bazaar, can allow tenants to achieve their cost goals too. We also discuss how exploiting malleability of more than two resources can be accomodated with Bazaar, and, as an example, provide a case-study exploiting malleability along the *time domain.* Our findings show that exploiting time malleability can further reduce median job completion time by more than 50%. Overall, we argue that the higher-level tenant-provider interface enabled by Bazaar benefits both entities. Tenants achieve their goals, while the resource selection flexibility improves datacenter goodput and hence, provider revenue.

On a broader note, the resource elasticity enabled by cloud computing can allow tenants to trade-off higher costs for better performance. Bazaar makes this trade-off explicit for MapReduce-like applications. We believe that if such elasticity of performance and costs were to be extended to a broader set of applications, it would remove a major hurdle to cloud adoption.

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 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 [21,16,17] migrating to the cloud. Data-parallel frameworks like MapReduce [22], Dryad [23], or Scope [19] 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 nodes 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 in a timely fashion. However, with today's setup, tenants are responsible for mapping such high-level completion time goals down to specific resources needed for their jobs [3,4]. This has led to a slew of proposals for optimization, mostly focusing on MapReduce– determining good configuration parameters [4], storage and compute resources [3], and better scheduling [24]. Recent efforts like Elasticiser [5] address the problem at a higher-level and strive to determine the number and type of VMs needed to ensure a MapReduce job achieves the desired completion time.

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 [6,7,8]. This, in turn, impacts application performance, irrespective of the number of assigned VMs. For instance, Schad et al. [7] 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 practically infeasible.* We show evidence of this in § 4.2.2.

At the same time, ignoring the tenant's requirements hurts providers as well. For instance, it results in poor VM placement and high contention across the internal datacenter network. This leads to poor performing jobs that prevent the provider from accepting subsequent tenant requests. Such outliers have been shown to drag down the system throughput and, hence, the provider revenue by as much as 60% [2].

Multi-resource elasticity. Performance issues with shared resources, such as the ones described above, have prompted a slew of proposals that offer guaranteed performance atop such resources [2,12,13,14,15]. With these, tenants can request resources beyond just VMs; for instance, [2,12,13] allow tenants to specify the network bandwidth between their VMs. We note that providing tenants with a guaranteed amount of individual resources makes the problem of achieving high-level performance goals tractable.

This paper exploits such multi-resource elasticity and builds upon efforts that provide guaranteed resources. *We consider a two resource tenant-provider interface whereby tenants can ask for VMs and internal network bandwidth.* As proposed in [2,12], a tenant request is 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, we first quantify its impact on typical cloud applications.

2.1 Malleability of data-analytics applications

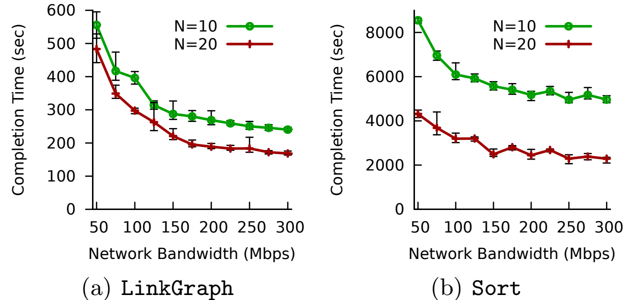
We first focus on data analytic frameworks, and use MapReduce as a running example. Our goal is to study how its performance is affected when varying different resources.

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 [25]. `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



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

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

detailed in §4.1, here we just focus on the performance trends.

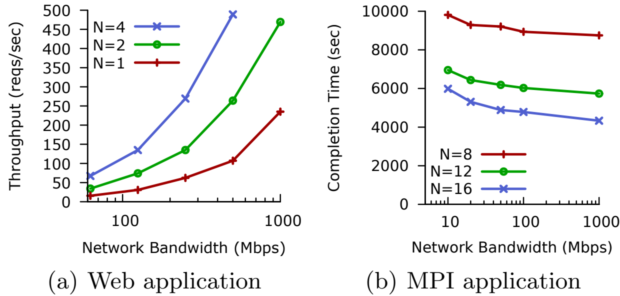
Figure 1(a) shows the completion time for `LinkGraph` on a cluster with 10 and 20 nodes and varying network bandwidth. As the bandwidth between the nodes increases, the time to shuffle the intermediate data between map and reduce tasks shrinks, and thus, the completion time reduces. However, the total 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 this value would still shrink the completion time. This was also confirmed by the experiments that we ran on our testbed (see Section 4.3) where we used in-memory data shuffle to avoid incurring the disk bottleneck.

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. While the results are detailed in Section 4.1 (Figures 6(a) and 7(a)), we find 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



(a) Web application (b) MPI application

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

cloud applications as well. We briefly discuss two such typical 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.

MPI application. We used an MPI application generating the Crout-LU decomposition of an input matrix as an example for cloud HPC and scientific workloads. Figure 2(b) shows the completion time for a 8000x8000 matrix with varying N and B. Given the processor-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 by 15-25% only.

Overall, the experiments above lead to two obvious yet key findings. First, *the performance of typical cloud applications depends on resources beyond just the number of compute instances*. Second, *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. Our evalaution later shows that this flexibility is important in the context of enabling predictable performance for tenants; 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 capitalize on resource malleability while satisfying tenant high-level performance goals. To this end, we identify two conditions that our target applications must satisfy:

(1). *Resource malleability.* The application performance should vary with two or more resources. Further, it should be possible to trade-off one resource for the other without impacting performance.

(2). *Performance predictability.* It should be feasible to predict the performance of the application when running on a given set of resources.

The results in this section suggest that the first condition holds for a fair fraction of cloud applications. To help ground our arguments, we hereon focus on MapReduce as an example cloud application. Based on this, we design Bazaar, a sytem that enables predictable performance for data analytic workloads in multi-tenants datacenters offering guaranteed VM and network resources. 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 [16,17] and (ii). it satisfies both our conditions very well. Its performance scales with the two resources we consider (condition 1) and as we show later, the well-defined architecture of MapReduce lends itself well for automatic analysis and low-overhead profiling (condition 2). However, as discussed in Section 5, *the core ideas in this paper can be extended both to applications beyond MapReduce and beyond two resources.* This is especially true as performance prediction for other cloud applications (eg., web [26] and ERP applications [27]) could be used instead of our MapReduce profiling tool without affecting Bazaar's overall operation.

3. Bazaar DESIGN

Figure 3 shows Bazaar's design model. Tenants submit the characteristics of their job, and high-level requirements such as the job completion time and/or desired

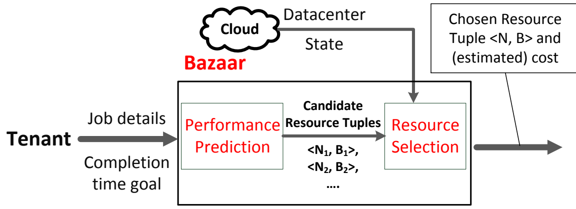


Figure 3: Bazaar design model.

cost to Bazaar. In the context of MapReduce, the job characteristics include the MapReduce program, input data size and a representative sample of the input data. Bazaar translates tenant requirements to multiple *resource tuples* $<N, B>$, each comprising the number of VMs and the network bandwidth between the VMs. Using the current state of the datacenter, Bazaar selects the resource tuple that is most amenable for the provider and the job cost. Note that this description assumes that Bazaar operates as a provider-side tool; yet, this is not necessary. Bazaar can also be implemented as a third-party service with appropriate interfaces to the cloud provider to obtain the required information.

As shown in the figure, Bazaar translates tenant goals into resource tuples using 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.

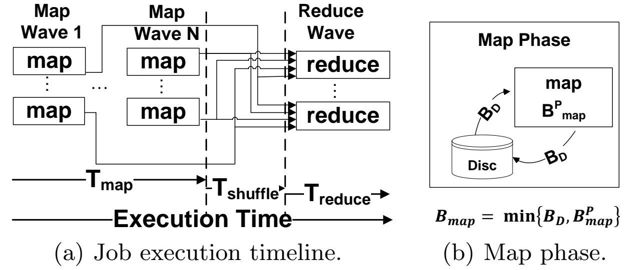
(2). A *resource selection* component that selects *which* resource tuple should be allocated. This choice is made so as to minimize the impact of the request on the provider's ability to accommodate future tenant requests.

The following subsections describe these two components in detail.

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 [28], user software [29], databases [30]. In the context of MapReduce, efforts like Mumak [31] and MRPerf [32] have built detailed MapReduce simulators that can be used for prediction. However, this results in significant complexity and non-trivial prediction times. To allow for an exploration of different resource combinations, Bazaar requires fast yet reasonably accurate prediction. Inspired by profiling-based approaches for fast database query optimization [30], we capitalize on the well-defined nature of the MapReduce framework to achieve faster and simpler prediction for MapReduce jobs.

We design a prediction tool called *MRCute* or MapReduce Completion Time Estimator. MRCute takes a gray-box approach to performance prediction by complementing an analytical model with job profiling. We first develop a high-level model of the operation of MapReduce jobs and construct an analytical expression for a job's completion time (white-box analysis). The resulting 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 (black-box analysis).



(a) Job execution timeline. (b) Map phase.

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

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 and profiling

As shown in Figure 4(a), the execution of MapReduce jobs comprises 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 $T_{map} = W_{map} * \frac{I_{map}}{B_{map}}$, 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 MapReduce job, we develop an anlytical model by applying bottleneck analysis [33] to the MapReduce framework.

For example, during the **map phase** (Figure 4(b)), each map task reads its input off the local disk, 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 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). To simplify exposition, the complete description of the analytical model is provided in the Appendix.

Besides the input parameters specified in eq. 1, our analytical model 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.

The job profiler assumes that the phase bandwidth observed during profiling is representative of actual job operation. Satisfying this assumption poses two challenges:

1). Infrastructure heterogeneity. Ideally, 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 [34]. Indeed, we observed variable disk performance even during our small-scale experiments which significantly degrades prediction performance; to counter this, MRCute maintains statistics regarding the disk bandwidth of individual machines (see §4.1).

2). Representative sample data. The sample data used for profiling should be representative and of sufficient size. If 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 (see §4.1.1).

3.1.2 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 10-20) 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.

3.2 Resource selection

Since all the candidate tuples for a specific job achieve similar completion times, the provider has the flexibility regarding 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. 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. For instance, 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 their network links is 400 Mbps (2*200 Mbps).

To address this, we use the 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. Hence, 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 VMs that have been allocated.

Overall, given the set of utilization vectors corresponding to the feasible tuples, we are interested in minimizing the number of rejected requests in the future. This problem has been studied in various contexts, such as online ad allocation [35], online routing and admission control in virtual circuit networks [36], etc. Depending on the context, one can show that different cost functions (that measure the cost for accepting a request or allocation) yield optimal scheduling for different resource allocation models [37]. We experimented with a number of such cost functions and found that associating a feasible tuple with a cost function reflecting the resource imbalance it causes performs significantly better in terms of minimizing rejected requests. In our setting, resource imbalance translates to choosing the utilization vector that balances the capacity left on resources after the request has been allocated. Precisely, our heuristic aims to optimize the following across all resources

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

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 [38]. An extra complication in our setting is that the hierarchical nature of typical datacenters implies that there is a hierarchical set of resources corresponding to datacenter hosts, racks and pods. Below we describe in detail how this heuristic is extended to our scenario.

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.

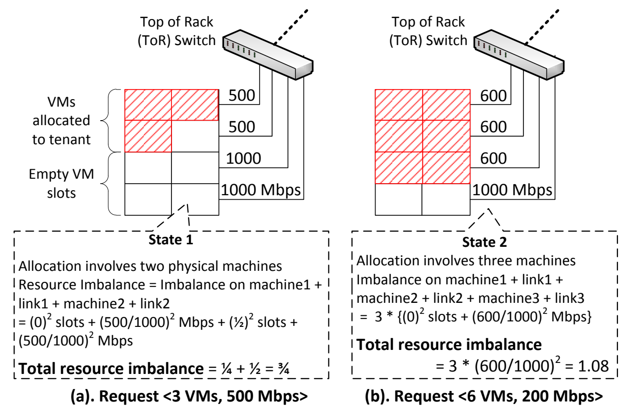


Figure 5: Selecting amongst two feasible resource tuples. Each physical machine has 2 VM slots and an outbound link of capacity 1000 Mbps. Each link is annotated with its residual bandwidth.

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 in a datacenter with 2 VM

slots and a Gigabit link per machine. 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.

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 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).

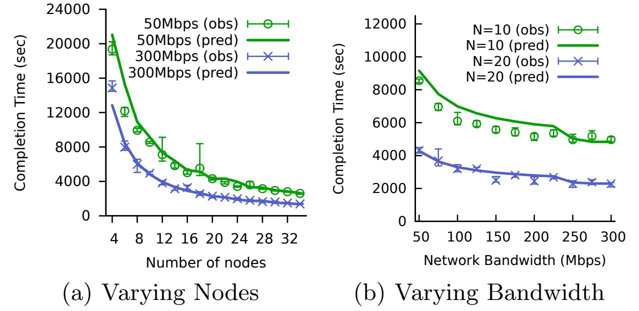
(2). We use large scale simulations to evaluate the benefits of Bazaar. Capitalizing on resource malleability significantly improves datacenter goodput (§4.2).

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

4.1 Performance prediction

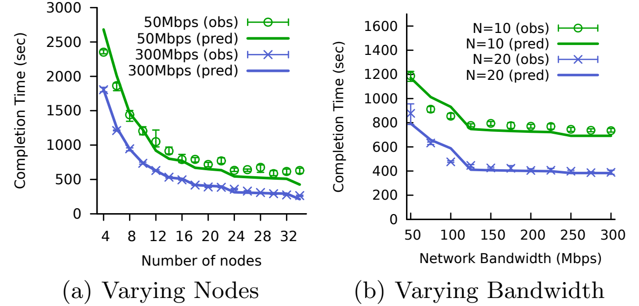
We use MRCute to predict the job completion of the five MapReduce jobs described in §2.1 (Table 1). For each job, MRCute predicts the completion time for varying number of nodes (**N**) and the network bandwidth between them (**B**). As detailed in §3.1, the prediction involves profiling the job with 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. Each node has a quad-core Intel Xeon 2.4GHz processor, 12 GB RAM



(a) Varying Nodes (b) Varying Bandwidth

Figure 6: Predicted completion time for `Sort`, an I/O intensive job, matches the observed time.



(a) Varying Nodes (b) Varying Bandwidth

Figure 7: Predicted completion time for `WordCount`, a processing intensive job, matches the observed time.

and a 1Gbps 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 [5], 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. In the next paragraphs, we examine in detail the prediction results of the various jobs, and the factors introducing errors in our model.

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 and 7 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 95th percentile.

To understand the root cause of the prediction errors, we look at the per-phase completion time. Figure 8 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

Hadoop Job	Stages	Sample Data Size	Profiling Time	Prediction error (all runs)		Prediction error (B=300Mbps)	
				Average	95%ile	Average	95%ile
Sort	1	1GB	100.8s	8.9%	20.5%	4.56%	6.56%
WordCount	1	450MB	67.5s	8.4%	19.7%	4.19%	12.34%
Gridmix	3	16GB	546s	11.5%	17.8%	15.2%	20.05%
TF-IDF	3	3GB	335s	5.6%	9.7%	3.45%	4.75%
LinkGraph	4	3GB	554.8s	8.2%	12.3%	5.15%	9.8%

Table 3: Profiling information and the prediction error of MRCute for various Hadoop jobs

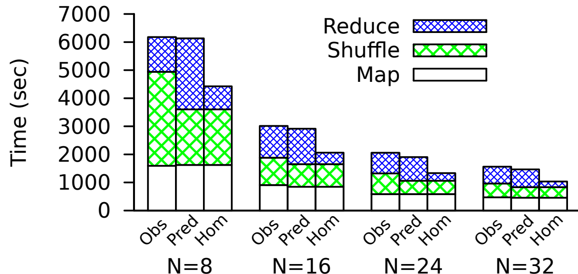


Figure 8: Per-phase breakdown of the observed (*Obs*) and predicted (*Pred*) completion time for `Sort` with bandwidth = 300 Mbps. *Hom* represents the predicted time assuming homogeneous disk performance.

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 straggling map tasks 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 straggling reduce tasks resulting from failures, uneven data distribution, or other factors, inflate the phase and in turn, the job completion time. Since we opted for simplicity and fast prediction times, our model does not account for stragglers beyond those resulting from disk performance. Overall, such straggler introduce errors in the predicted completion time.

Nevertheless, even though we do not model task failures, in some cases the predicted completion time can still be accurate. For example, the discrepancy in the shuffle and reduce phase times at 8 nodes in Figure 8 is due to failures. During the job execution, some reduce tasks failed and were restarted by Hadoop. The restarted reduce tasks begin their network transfers later than other reducers and this extends the actual shuffle phase time. However, the reduce phase time shrinks since the restarted reduce tasks have less disk contention. Overall, the shuffle phase lasts longer, the reduce phase is shorter and hence, despite the failed tasks, the completion time stays the same.

Benchmarking disk performance. While we can get away with not modeling failures, variability in disk performance cannot be ignored. To highlight the importance of benchmarking individual disks for their I/O bandwidth, the bars in figure 8 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 I/O 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. To account for this variability, MRCute maintains statistics regarding the disk bandwidth of individual machines in the datacenter. In practice, this can be obtained by profiling the machines periodically, for instance, when they are not allocated to tenants.

Beyond `Sort` and `WordCount`, the predicted estimates for the other two jobs show similar trends. For brevity, we summarize the prediction errors in Table 3. Overall, we find a maximum average error of 11.5% and a 95th percentile of 20.5% across all runs (resp. 15.2% and 20.05% when B is 300Mbps). Apart from `Gridmix`, the error is lower when the network bandwidth is 300Mbps or higher. This is due to the poor I/O bandwidth offered by the disk on the testbed nodes. At this range, the network stops being the bottleneck and the completion time is dictated by the disk and CPU.

4.1.1 *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 Table 3. 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 [39]. `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.* Table 3 also shows the time to profile individual jobs. For `Sort` and `WordCount`, the

profiling takes 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 expect typical jobs to last at least a few hour.

To summarize, these experiments indicate that MRCute can indeed generate good completion time estimates for MapReduce jobs. While the prediction accuracy could be improved with more detailed modeling and profiling, this would come at expense of higher estimation time. Fast and low overhead prediction is important for our setup. We expect SLAs for data analytic jobs to allow for some variance ($\pm 10\%$). Alternatively, Bazaar can account for such prediction errors by actually estimating the resources required to complete a tenant job in, say, 90% of the desired completion time. Hence, the prediction error can be well accommodated.

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. By varying the connectivity and the bandwidth of the links between the switches, we vary the oversubscription of the physical network. 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 [40], 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 $B^{\mathcal{P}}_{reduce}$, while a processor intensive job like `WordCount` has low values. 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 requested from the cloud provider who allocates these VMs on the datacenter using the allocation algorithm in [2]. *If the provider cannot allocate the request because of insufficient resources, the request 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 with the program parameters described above and the disk bandwidth. 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.

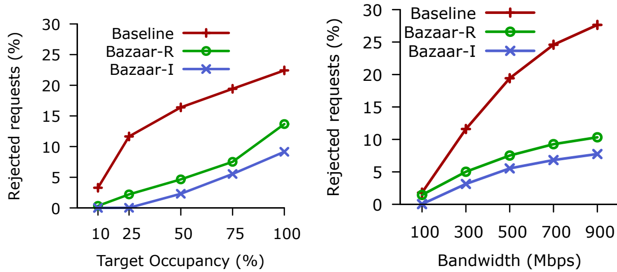
Resource selection strategies. We evaluate three strategies to select a resource tuple.

(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 the selection of a tuple bears on the provider.

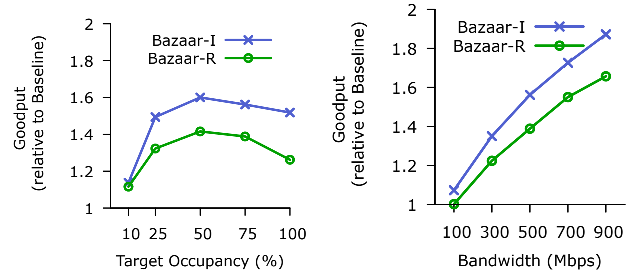
(3). *Bazaar-I* (imbalance-based selection). For each tuple, we determine how it would be allocated and calculate the resulting utilization vector and resource imbalance. The tuple with the lowest resource imbalance is chosen.

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 targetted occupancy on a datacenter with M total VMs is $\frac{\lambda N T}{M}$, where T is the mean completion time for the requests



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

Figure 9: Percentage of rejected requests, varying mean bandwidth and target occupancy.



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

Figure 10: Datacenter goodput with varying mean bandwidth and varying target occupancy.

and N is the mean number of requested VMs in the Baseline scenario.

Simulation breadth. The VM-level interface offered by today's cloud providers entails that most measurement studies focus on the number of VMs (or in general, machines) used by tenant jobs. Given the lack of information on requirements of typical tenants, our evaluation explores the entire space for most parameters; these include the resources needed to achieve tenant goals, target occupancy, and physical topology oversubscription. This is not only useful for completeness, but further provides evidence of Bazaar's performance at the extreme points.

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 [41].

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 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 9(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 9(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 10 shows the datacenter goodput for the Bazaar selection strategies relative to Baseline. Depending on the occupancy and bandwidth, *Bazaar-I improves the throughput 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 10(a), the relative improvement in goodput with Bazaar strategies first increases with target occupancy and then declines. This pattern emerges as at both low and high occupancy, there is not as much room for improvement. Requests either arrive far apart in time and most can also be accepted by Baseline, or in the other extreme, the arrival rate is high and

the datacenter is heavily utilized. In figure 10(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 map and reduce bandwidth below.

Figure 11 shows the relative goodput with varying network oversubscription. *Even in a network with no oversubscription* [43,44], *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 (2% over baseline). Thus, there are very few candidate resource tuples and the gains with Bazaar are small. However, as the disk bandwidth improves, there are more candidate tuples to choose from and, hence, the performance of Bazaar 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 9 and 10.

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 12(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 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 the statistical multiplexing, it obtains a higher utilization. Yet, for high values of occupancy, which are typical of today's datacenters [42], the performance of Fair-sharing drops significantly. 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 Figures 12(b) and 12(c), which show that i) rejected requests significantly increase, and ii) job completion time (and hence tenant cost) is extended for at least 50% of the jobs due to network contention and for 12% of the jobs the actual completion time is at least twice the desired completion time.

4.3 Deployment

We complement our simulation analysis with experiments on a small-scale testbed using a prototype implementation of Bazaar. The testbed consists of 26 Dell Precision T3500 servers with a quad core Intel Xeon 2.27GHz processor and 12 GB RAM, running Windows Server 2008 R2. Servers have one 1 Gbps Intel PRO/1000 NIC each and are all connected to the same switch.

The goal of these experiments is twofold. First, we want to demonstrate the *feasibility* of Bazaar, measuring the overhead introduced per request. Second, we want to *cross-validate* the accuracy of our simulator by comparing the results obtained in the testbed with the results obtained in the simulator when running the same workload.

Feasibility analysis. To evaluate the performance of our Bazaar prototype at scale, we measured the time to allocate tenant requests on a datacenter with 128,000 VMs. This includes both the time to generates 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 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.

Cross-validation. To validate the accuracy of our simulator, we ran scaled-down experiments on our testbed and replicated the same workload in the simulator. 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. As in Section 4.2.1, we do not execute the map and reduce phase but we do account for the time spent in each of the two phases using our prediction model. In the shuffle phase, we generate an all-to-all traffic matrix with an average input data size of 4.75 GB per job. We use the Windows Traffic Control API [45] on individual machines to enforce the rate limits. To remove disk bottlenecks, we generate all data to shuffle in memory.

Figure 13 compares the accepted requests and goodput across the simulator and testbed for Baseline and

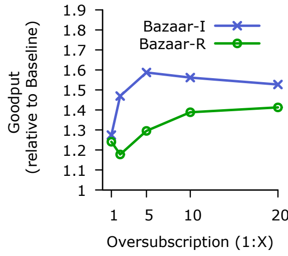
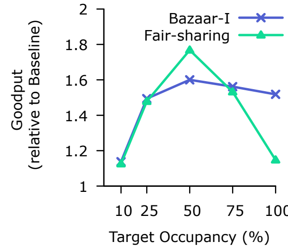
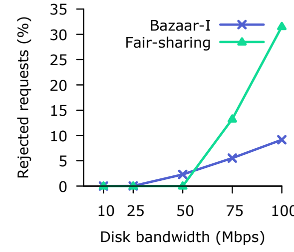


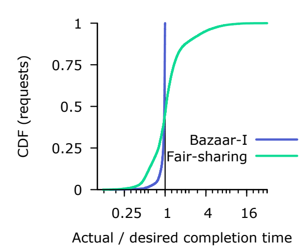
Figure 11: Network oversubscription (occupancy is 75%).



(a) Datacenter goodput.



(b) Rejected requests.



(c) CDF of completion time.

Figure 12: Comparing against today's setup (Mean BW is 500 Mbps).

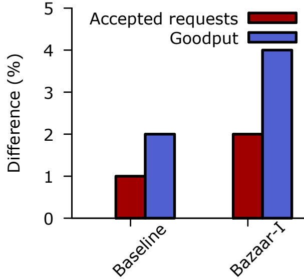


Figure 13: Cross-validation between simulation and testbed.

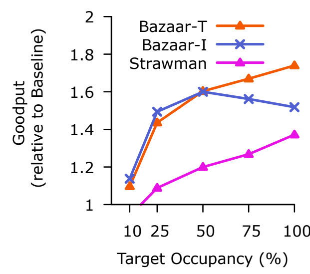


Figure 14: Datacenter goodput when exploiting time malleability.

Bazaar-I. Like in the previous experiments, we use a target occupancy of 75%, mean B_{base} of 500 Mbps with total requests scaled down to 500 and mean N_{base} equal to 7. In all cases, the maximum difference in the number of accepted requests and goodput is roughly 2%, and 4% respectively. We also rewired the servers to obtain an oversubscription ratio of 1:5, achieving similar results. Although our testbed reflects a small scale deployment, the benefits in the specific scenario relative to the Baseline were still 9% and 2.5% for Bazaar-I and Bazaar-R respectively. This provides further evidence that once the disk bottleneck from the shuffle phase is removed, resource malleability offers significant benefits.

4.4 Beyond two resources: time malleability

Our evaluation considered application malleability along two dimensions, i.e., N and B. However, the ideas presented throughout the paper can be extended to other resources, e.g., storage, or even exploit the malleability along the *time* axis. Here, we briefly explore this opportunity. The idea is that if additional resources (beyond those needed to accommodate a given tenant's request) are available, the provider can devote these additional resources to tenant jobs, so that the job completion time is reduced. In this way, the resources used by the job can be reclaimed earlier and, hence, 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 14 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 $\tilde{8}0\%$.* In Figure 14 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 over-provision the resources for the early requests, which reduces the ability to accommodate future ones.

5. DISCUSSION

Bazaar-T provides another example of how Bazaar 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. Such a *resource-based pricing* model could also be naively extended for a multi-resource setting. However, this results in a mismatch of tenant and provider interests. The cheapest $< N, B >$ resource tuple to achieve the tenant's goal may not concur with the provider's preferred resource combination. Further, resource based pricing entails tenants paying 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 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. Introducing such *job-based pricing* benefits 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.

Bazaar, when combined with job-based pricing, can 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. By serving as a conduit for exchange of information between tenants and providers, Bazaar provides benefits for both entities.

6. REFERENCES

[1] Michael Armburst et al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Tech. Rep., 2009.

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

[3] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues, "Conductor: Orchestrating the Clouds," in *LADIS*, 2010.

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

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

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

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

[8] "Measuring EC2 system performance," http://bit.ly/48Wui.

[9] A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Production Cloud Services," Delft University of Technology, Tech. Rep., 2010.

[10] M. Zaharia, A. Konwinski, A. D. Joseph, Y. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[11] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running HPC applications in public clouds," in *HPDC*, 2010.

[12] P. Soares, J. Santos, N. Tolia, and D. Guedes, "Gatekeeper: Distributed Rate Control for Virtualized Datacenters," HP Labs, Tech. Rep. HP-2010-151, 2010.

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

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

[16] "Hadoop Wiki: PoweredBy," http://goo.gl/Bbfu.

[17] "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce/.

[18] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data Warehousing and Analytics Infrastructure at Facebook," in *SIGMOD*, 2010.

[19] 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.

[20] "Amazon Cluster Compute," Jan. 2011, http://aws.amazon.com/ec2/hpc-applications/.

[21] "Big Data @ Foursquare ," http://goo.gl/FAmpz.

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

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

[24] 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.

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

[26] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "Webprophet: automating performance prediction for web services," in *NSDI*, 2010.

[27] D. Tertilt and H. Krcmar, "Generic Performance Prediction for ERP and SOA Applications," in *ECIS*, 2011.

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

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

[30] M. Kremer and J. Gryz, "A Survey of Query Optimization in Parallel Databases," York University, Tech. Rep., 1999.

[31] "Mumak: Map-Reduce Simulator," http://bit.ly/MoOax.

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

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

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

[35] 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.

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

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

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

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

[40] "Tom's Hardware Blog," http://bit.ly/rkjJwX.

[41] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Sharing the Datacenter Network," in *NSDI*, 2011.

[42] "Amazon's EC2 Generating 220M," http://bit.ly/8rZdu.

[43] 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.

[44] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[45] "Traffic Control API," http://bit.ly/nyzcLE.

[46] 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.

APPENDIX

1). Phase bandwidth. We described the model of the map phase in Section 3.1.1. Following the same logic for the **reduce phase**, $B_{reduce} = Min\{B_D, B^{\mathcal{P}}_{reduce}\}$. Dur-

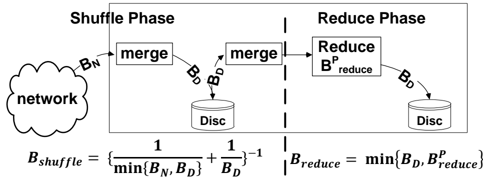


Figure 15: Detailed description of the resources involved in the shuffle and reduce phase.

ing the **shuffle phase**, reduce tasks complete two operations. Each reduce task first reads its partition of the intermediate 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 the final merge is performed in memory before the data is consumed by the reduce phase. This operation is bottlenecked at the disk, i.e., bandwidth $= B_D$. Given that the two operations occur in series, the shuffle phase bandwidth is

$$B_{shuffle} = \left\{ \frac{1}{Min\{B_D, B_N\}} + \frac{1}{B_D} \right\}^{-1}.$$

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

$$\begin{aligned} T_{estimate} &= T_{map} + T_{shuffle} + T_{reduce} \\ &= \lceil \frac{M}{N*M_c} \rceil \left\{ \frac{|I|/_M}{B_{map}} \right\} + \left\lceil \frac{R}{N*R_c} \right\rceil \left\{ \frac{|I|/_{\{S_{map}*R\}}}{B_{shuffle}} \right\} \\ &\quad + \left\lceil \frac{R}{N*R_c} \right\rceil * \left\{ \frac{|I|/_{\{S_{map}*S_{reduce}*R\}}}{B_{reduce}} \right\}. \end{aligned}$$

The previous discussion 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 [46]. 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}. Finally, we also assume that the set of keys is evenly distributed across reducers. In case of skewed key distribution, we need to sample the input to determine the worst-case reducer load.