Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters

Carlo Fuerst[1] Stefan Schmid[2] Lalith Suresh[1] Paolo Costa[3]
[1] TU Berlin, Germany [2] Aalborg University, Denmark [3] Microsoft Research
{carlo,lalith}@inet.tu-berlin.de schmiste@cs.aau.dk paolo.costa@microsoft.com

***Abstract*—In multi-tenant cloud environments, the absence of strict network performance guarantees leads to unpredictable job execution times. To address this issue, recently there have been several proposals on how to provide guaranteed network performance. These proposals, however, rely on computing resource reservation schedules *a priori*. Unfortunately, this is not practical in today's cloud environments, where application demands are inherently unpredictable, e.g., due to differences in the input datasets or phenomena such as failures and stragglers.**

To overcome these limitations, we designed KRAKEN, a system that allows tenants to dynamically request and *update* minimum guarantees for both network bandwidth and compute resources *at runtime*. Unlike previous work, Kraken does not require prior knowledge about the resource needs of the tenants' applications but allows tenants to modify their reservation at runtime. Kraken achieves this through an *online resource reservation scheme* which comes with provable optimality guarantees.

In this paper, we motivate the need for dynamic resource reservation schemes, present how this is provided by Kraken, and evaluate Kraken via extensive simulations.

I. INTRODUCTION

Cloud-based applications, including batch processing, streaming, and scale-out databases, generate a significant amount of network traffic and a considerable fraction of their runtime is due to network activity. For example, traces of jobs from a Facebook cluster reveal that network transfers on average account for 33% of the execution time [16].

Unfortunately, as reported in previous studies [5], in existing cloud infrastructures the bandwidth available to the tenants varies significantly over time, i.e., by a factor of five or more [25], even within the same day. Given the time spent in network activity by these applications, this variability has a non-negligible impact on the application performance, which makes it impossible for tenants to accurately estimate the execution time in advance [17].

Over the last years, several solutions have been proposed to improve the sharing of network bandwidth among tenants, by leveraging admission control and bandwidth reservations, thus enabling tenants to specify *absolute* guarantees [5], [7], [11], [14], [20], [21], [23]. In particular, many of these proposals offer a *virtual cluster* abstraction [5], [7], which provides the tenants with the illusion of having their own dedicated network. A virtual cluster *guarantees* a specified *minimal* bandwidth between all tenant's virtual machines, independently of their locations in the datacenter topology.

However, the vast majority of existing solutions providing absolute bandwidth guarantees are based on *offline* and *constant* reservations schemes [5], [7], [11], [14], [19], [23]: they require that tenants announce the entire resource reservation schedule *ahead of time*, i.e., at job submission time. They typically assume that the corresponding resource reservations need to be *constant* over time, and hence tenants either have to over-provision during idle times (thus reducing efficiency and inflating cost) or under-provision during peak times (thus reducing application performance), or both. Noteable exceptions are Cicada [15], which offers predicitive instead of absolute guarantees, and Proteus [7], which allows tenants to specify time-varying bandwidth reservations. However, even with Proteus, the reservations must be made at the startup time and they cannot be changed afterwards. This inflexibility is at odds with the cloud computing paradigm, which enables elasticity by allowing tenants to "scale out" or "scale in" their applications at runtime. We argue that in most cases it is very hard to accurately estimate application resource needs ahead of time, rendering offline reservation schemes inadequate. Several factors contribute to this unpredictability including unexpected events such as stragglers and failures [3], [4] as well as spikes in application demand (flash crowds).

A naive approach to enable runtime reconfiguration would be to restart the resource allocation from scratch every time an update request is received from the tenants. This, however, would introduce an unacceptable overhead as most (if not all) the compute resources such as VMs need be migrated. At the other extreme, there are approaches such as Blender [23] that support a weak form of reconfiguration by allowing tenants to update rate limiters at runtime. This, however, prevents users from upgrading both compute and network resources at the same time. More importantly, as we show in the evaluation, since no migration is considered, the efficacy of the solution is very limited. In this paper, we strike a balance between these two approaches by allowing users to dynamically reconfigure both compute and network resources *simultaneously* while minimizing the number of migrations.

Our Contribution. We make the following contributions.

1) *The need for online resource reservation schemes:* We show that offline resource reservation schemes are insufficient: Even for simple Hadoop jobs small internal changes can lead to significantly different executions. Therefore, in order to meet application performance goals, not only strict resource isolation needs to be provided, but also a possibility to update these reservations at runtime.
2) *The Kraken system:* We design Kraken, a system which supports the online (and joint) update of both bandwidth

as well as the compute resources. Kraken can also perform migrations in order to satisfy upgrade requests: While the migration of entire virtual machines may be expensive in practice, Kraken only assumes that compute units, the endpoints of traffic flows, can be migrated. Kraken comes with provable performance guarantees and ensures (*i*) the satisfaction of all upgrade/downgrade requests for which this is possible, (*ii*) minimal reconfiguration and resource costs, (*iii*) low runtimes.

3) *Benefits of online resource reservations:* Our simulations show the benefits of elastic resource reservations.

Kraken can be used for many applications that benefit from resource elasticity, including batch-processing applications (e.g., graph processing or distributed databases) or high-performance computing applications.

Non-Goals. We focus on *how* to efficiently embed and reconfigure virtual clusters; a detailed discussion of *when* to change a virtual network specification is left for future work. The time and extent of upgrades and downgrades depend on the setting, on the type of application, as well as on the tenants' objectives. In this paper, we advocate for a clean interface between tenant and provider over which such reconfiguration requests can be issued based on the tenants' needs.

II. Motivation for an Online Approach

Before presenting our solution in detail, we argue that today's offline reservation schemes are not sufficient to ensure application performance guarantees in an efficient manner.

We distinguish between two offline reservation schemes: (1) schemes with *constant* resource reservations such as the ones proposed in [5], [11]; and (2) schemes such as Proteus [7] with time-varying resource reservations which need to be announced ahead of time and, hence, require accurately predicting a job's resource-utilization over time, e.g., using data from previous runs.

Constant reservation schemes are wasteful for any application with time-varying resource demands, such as MapReduce applications, which cycle between network-intensive and compute-intensive phases [7], or an online computer game whose demand is subject to time-of-day effects [25].

While offline and time-varying reservations may be possible in idealized conditions, in practice, this is rarely the case. This is obvious for continuously running applications, such as a web-service or video-on-demand service, whose popularity can change significantly and unexpectedly. But, as we show next, even the resource pattern of very simple MapReduce applications are hard to predict accurately. It has been reported that stragglers can be several times slower than the median task completion time [3], [4], [8], [13], [24]. Stragglers occur due to a variety of environmental factors such as slow disks and failures. Cluster frameworks typically use control-loops based on these factors to (re-)schedule tasks, e.g., Hadoop's speculative executor. This makes it hard to predict if there will be stragglers in the first place and if so, when and where the cluster framework will re-schedule a slow task.

To highlight this we perform a simple and idealized experiment wherein we run a single Hadoop cluster in an OpenStack-based testbed. For this we use five physical servers (8 CPU cores and 64GB of RAM) with one virtual machine each. Each virtual machine is allocated 4 virtual cores with 4 GB of RAM. Each node is mapped to a virtual machine each (one master, four slaves). The workload consists of a TeraSort job, operating on 150 million 100-byte records.

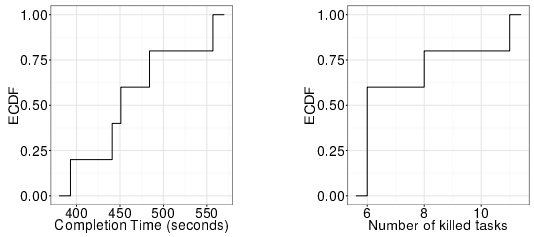

Fig. 1: Execution unpredictability—Completion times of jobs in the presence of speculative execution (*left*) and the number of speculated tasks (*right*).

We repeat the experiment five times with speculative execution enabled. Figure 1 (*left*) indicates the variance in job completion times across the runs: a range of 150 seconds. This observation is also supported by Figure 1 (*right*) which indicates the number of straggling tasks that were speculatively re-executed by the Hadoop cluster.

Note that since TeraSort is IO-bound and all data are randomly generated with a uniform distribution, its behavior is much more regular than most other jobs used in data analytics, which can suffer from skewed data distribution, irregular computation patterns, etc. Therefore, we expect real jobs to exhibit even higher variance across runs, as it is often reported in literature [3], [4], [8], [13]. These observations serve to demonstrate that even with the same workload and a dataset of the same size being re-executed, it is difficult to predict how a job progresses over time.

In conclusion, we argue that offline approaches for resource reservations such as Proteus do not suffice, as cloud environments such as Amazon EC2 [24] are even more noisy than our environment studied here. This makes it more difficult to predict performance of an application a priori, which underlines the need for dynamic and online reservation schemes.

III. Model & Example

We start by introducing the settings and the virtual network abstraction considered in this paper, and subsequently highlight the algorithmic challenge.

A. Setting

We consider the standard *Virtual Cluster* abstraction to model virtual networks with strict performance guarantees [5], [7], [17]. A virtual cluster offers the tenant the illusion for all her *Compute Units (CUs)* to be attached to a single non-oversubscribed switch with a minimum bandwidth b guaranteed. If excess bandwidth is available, it can be used in addition to the reserved bandwidth, e.g., using recently proposed extensions to TCP such as Seawall [22].

A virtual cluster $\mathrm{VC}(n,b)$ has two parameters: n, the number of (identical) CUs in the cluster, and b, the bandwidth

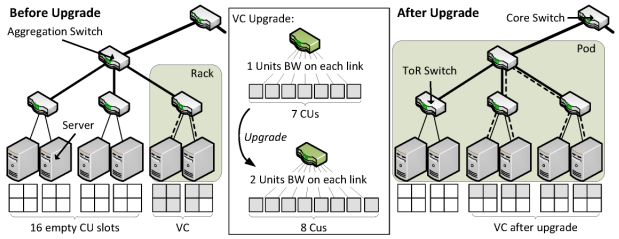

Fig. 2: Upgrade of a virtual cluster VC: *Left - the initial state:* VC(7,1) is embedded on the right-most rack of a pod of the fat-tree. The *dashed lines* indicate the current bandwidth reservations. *Middle - the upgrade request:* VC(7,1) needs to be upgraded to VC(8,2). *Right - after the upgrade:* Three CUs were migrated in order to find a new feasible embedding of VC which does not violate the capacity on the servers' uplinks.

reservation from each CU to the virtual switch. Virtual clusters belonging to different tenants need to be embedded on a given *substrate*: a physical network connecting a set of servers. In this paper, we focus on multi-rooted tree (or *fat-tree*) like physical network topologies [1], [10] as they are the predominant topology in today's data centers. These topologies are hierarchical and are recursively made of subtrees at each level. A fat-tree consists of a set of *pods* which are interconnected by *core routers*. Pods comprise a set of *racks* which are interconnected by the *aggregation switch*, and racks comprise multiple *servers* (or *hosts*) which are interconnected by the *Top-of-Rack (ToR) switch*. Each server can host a fixed number of CUs. As done in previous work, e.g., [5], [7], given the amount of multiplexing and assuming the availability of a multi-path routing protocol such as ECMP, we can approximate these links as a single aggregate link for bandwidth reservations.

To save costs, some data center operators introduce some degree of over-subscription, typically at the higher levels of the hierarchy. We model these configurations with two parameters $\gamma_1, \gamma_2 \geq 1$ (called the *over-subscription factors* in [5]): γ_1 denotes the factor of reduced capacity on the aggregation network (between ToR and aggregation switches) and γ_2 the factor of reduced capacity between the aggregation switches and the core switch.

The embedding of a virtual cluster describes its resource allocation in the substrate: an embedding maps each CU of the virtual cluster to a physical server in the substrate network; multiple CUs may be hosted on the same server. In addition, the embedding specifies the amount of bandwidth on each link reserved for the tenant. Intuitively, a "valid" embedding is one that does not oversubscribe server or network resources. A "good" embedding additionally chooses servers that are close in the physical network, thus minimizing unnecessary resource reservations on the physical links.

B. The Challenge

The goal of this paper is to support virtual clusters whose guarantees can be adjusted over time, in an online fashion. Specifically, we want to be able to (1) upgrade a virtual cluster VC(n,b) consisting of n CUs and with a bandwidth guarantee b, both in *size* (i.e., number of CUs) as well as in the minimum *bandwidth*, that is, to a virtual cluster with $x \geq 0$ more nodes and a factor $\delta \geq 1$ more bandwidth, i.e., to VC($n+x$,$b \cdot \delta$); (2) downgrade a virtual cluster in both size and bandwidth; (3) or a combination of both (e.g., upgrade size and downgrade bandwidth).

How to support such reconfigurations is also an algorithmic problem. Ideally, new feasible embeddings should be efficiently computable, i.e., at low runtime; moreover, we would like to avoid or at least minimize migrations in order to satisfy a reconfiguration request; finally, the resulting embeddings should have small network footprints, in the sense that no unnecessary bandwidth is reserved (on substrate links) to implement the virtual cluster guarantees.

C. Example

To illustrate both the model and the challenge, let us consider an example. Figure 2 (*left*) shows a part of a fat-tree, i.e., a single pod consisting of three racks with two servers each; each server has 4 CU slots. We assume that the uplinks of the servers have a capacity of 4 units and the fat-tree provides full bisection bandwidth ($\gamma_1 = \gamma_2 = 1$), resulting in a capacity of 8 units on the ToR switches' uplinks and a capacity of 24 units on the links between the aggregation switches and the core switch. On the right most rack, currently a virtual cluster VC is embedded; the dashed line indicates the path along with bandwidth is reserved to connect the CUs. At some point, VC is upgraded, from VC(7,1) to VC(8,2), see Figure 2 (*middle*).

How can this request be satisfied? Theoretically, the right server in the rack still has a free CU slot which could be used to accommodate the additional CU; however, doubling the bandwidth reservations for each the CUs will violate the bandwidth capacities on the uplinks of the servers. Hence it becomes necessary to distribute the CUs in the substrate, in order to reduce the bandwidth utilization of the uplinks of the two servers. Thus, in this scenario, some CUs need to be migrated to satisfy the request. Figure 2 (*right*) shows a solution: the resulting embedding is valid.

IV. The System

In this section, we first formalize the goals of the developed system, and then introduce the main concepts underlying Kraken and describe its key components.

A. Objectives

Kraken is designed to accept and implement any embedding and upgrade request whenever there are sufficient resources available in the substrate. Downgrade requests, instead, can always be satisfied.

Besides satisfying upgrade requests whenever this is possible, Kraken is designed (1) to optimize the embedding cost of the virtual cluster, i.e., the amount of bandwidth which needs to be reserved in the physical network to host the virtual cluster; and (2) to reconfigure existing embeddings locally, i.e., to minimize the migration cost. To avoid affecting the performance of other tenants, we do not allow the migration

of CUs belonging to other tenants, although in some cases this might result in lower embedding costs. The standard metric to evaluate the embedding cost (see also [5], [7]), is to measure the *embedding footprint* $F(\mathrm{VC})$ of a virtual cluster VC: $F(\mathrm{VC})$ is given by the overall network resources consumed by the VC, i.e., the sum of bandwidth reservations over all substrate links. (Note that the number of used CU slots is independent of the embedding.)

In order to measure the *reconfiguration costs*, we count the number of CUs which need to be embedded to a different location during an upgrade.

Notice that there is a trade-off between the two metrics: sometimes, at the price of higher reconfiguration costs, smaller footprints can be realized. In the following, we design our algorithms according to the following priorities (cf Section IV-F for a discussion of alternative objectives supported by Kraken): (1) the top priority is to satisfy a reconfiguration request; (2) the second priority is to minimize reconfiguration costs; and (3) the third priority, is to minimize the embedding footprint, i.e., among all solutions of the same reconfiguration costs, we compute the most resource efficient embedding.

Kraken provides the following worst-case guarantees.

1) *Request Satisfiability:* As long as a feasible solution exists all upgrade and downgrade requests are satisfied.
2) *Minimal Reconfiguration:* The reconfiguration cost is always minimized. In particular, if a solution without migrations exists, it is used. CUs of other tenants are never migrated.
3) *Optimal Allocation:* Among all possible solutions with minimal reconfiguration costs, Kraken computes the one with the minimal embedding footprint.
4) *Complexity:* The time complexity of re-configuring (or embedding) a virtual cluster is linear in the substrate size, in the worst-case.

B. Algorithmic Concepts

At the heart of Kraken lie two main concepts: (1) The *center-of-gravity* (or simply: *center*) of a virtual cluster and (2) the *slotCount values*. The center-of-gravity concept (introduced in [21]) allows us to decouple the embedding of the individual Compute Units (CUs), in the sense that, given the location of the center-of-gravity, the CUs can be mapped "greedily", one after the other, avoiding the combinatorial complexity and rendering the problem polynomial time solvable. The *slotCount*(v) values provide an aggregate information about the number of available CU slots in the subtree of the fat-tree below a given node v; they constitute the main data structure used by Kraken. While previous virtual cluster embedding algorithms used a similar concept [5], [7], [9], only the combination with the center-of-gravity concept allows a modification which enables the low runtime of the dynamic algorithm (roughly linear in the substrate size).

Center-of-Gravity. The virtual cluster abstraction offers tenants a network where each CU is connected to a virtual switch at bandwidth b [5]. While this virtual switch is only a logical concept, its position in the substrate matters, as resources need to be reserved from it to each CU.[1] The center-of-gravity may also be located on a server, not only on a switch (e.g., if many CUs of the virtual cluster are collocated on the same server). Given a mapping of the CUs of a given virtual cluster VC, we will refer to the optimal position of the virtual switch (with respect to embedding footprint) as the *center-of-gravity* CoG of VC.

Algorithm 1 Algorithm `upgrade`(VC,x,δ)

Output: success or *failure*

```
1: for all nodes v in the fat-tree: compute slotCount(v) values
2: m* ← ∞; F* ← ∞; cog* ← ⊥;
3: for all v in substrate do
4:    M ← minMig(v)
5:    if |M| ≤ m* then
6:       F ← footprint(v,|M|)
7:       if F < ∞ ∧ (|M| < m* ∨ F < F*) then
8:          cog* ← v
9:          m* ← |M|
10:         F* ← F
11:      end if
12:   end if
13: end for
14: if m* = ∞ then
15:   return failure
16: end if
17: μ ← computeEmbedding(VC,cog*)
18: return success
```

Given any node v in the fat-tree (either a server or a switch), we can partition the nodes of VC into two sets with respect to v: the set of CUs *at or below* the node v in the fat-tree, and the remaining CUs *above* (or "outside") v. Sometimes, we use the same terminology to refer to the location of substrate components relative to each other.

When applying the CoG concept to the fat-tree topology, we have two important properties, which Kraken leverages: (1) no more than half of the nodes, can be *above* CoG and (2) no more than half of the nodes are *below* one of the children of CoG. The correctness of this property can be shown easily by contradiction: If more than half of the CUs are behind one link, moving the CoG in this direction will decrease the bandwdith costs for more than half of the CUs by 1 and increase the costs for the other CUs by 1, resulting in a smaller footprint.

Moreover, when computing the embedding footprint of a virtual cluster VC, it is often helpful to count the number of CUs which are embedded *below* CoG(VC); we will refer to this number as β. The remaining CUs of VC which are embedded *above* CoG(VC), fall into three classes: the $\alpha^{(p)}$ "far-away" CUs located in a different pod, the $\alpha^{(r)}$ CUs in the same pod but in a different rack, and the $\alpha^{(s)}$ CUs in the same rack but on a different server. This classification results in simple formulas for the embedding footprint of a virtual cluster. For instance, if CoG(VC) is embedded to a top-of-rack switch, the embedding footprint is given by $F(\mathrm{VC}) = \beta + 3 \cdot \alpha^{(r)} + 5 \cdot \alpha^{(p)}$ as the distance to servers in the same rack (β) is 1 and the distance to all servers in the

[1] Note that there could be multiple positions with the same embedding cost, and that in a fat-tree, a distributed switch mapping does not reduce costs.

same pod but in diffrent racks ($\alpha^{(r)}$) is 3 while the distance to servers in other pods ($\alpha^{(p)}$) is 5.

***slotCount*-Values.** The second core concept of Kraken is the *slotCount*(v)-*value*: intuitively, the *slotCount*(v)-value indicates how many additional CUs can be placed below a certain substrate node v (a server or switch), such that the currently available server and link resources are all satisfied.

The number of CUs which can be placed below a certain substrate node v depends on two factors: the available bandwidth and the available CU slots. For Kraken it is sufficient to compute the bandwidth criteria for cases where CoG is *above* v. This eases the computation of these values significantly, since the resulting interval of possible amounts of CUs becomes continuous. In order to keep the runtime of the *slotCount* computation low, we leverage the optimal sub-problem property in our dynamic program: We start by computing the *slotCount*-values on the host level. For each server s we compute $slotCount(s) = \min(spareCUs(s), \lfloor spareBW(s)\ /b \rfloor)$ where $spareCUs(s)$ denotes the available CU slots of a server s and $spareBW(s)$ denotes the available bandwidth on the uplink. The *slotCount* of a rack r is then defined as: $slotCount(r) = \min(\sum_{s \in r} slotCount(s), \lfloor spareBW(s)\ /b \rfloor)$. The *slotCount*($p$)-values for pods can subsequently be computed from the racks' *slotCount*-values.

Overview. Based on these concepts, in order to embed or reconfigure a virtual cluster VC, Kraken simply cycles through all possible center-of-gravity locations in the substrate network (servers and switches): for each possible CoG location v, Kraken determines the minimal number of migrations needed, in order to shift the center to v. This is a fast operation since it does not scale with the size of the substrate, but with the size of the VC. If CoG can be implemented on v with minimal migration costs, the *slotCount* values are used to calculate the best possible embedding footprint of a mapping with the center at v. As we will show, this also does not require scanning the entire substrate, and is fast.

C. Upgrade Algorithm

Algorithm 1 shows the pseudo-code of Kraken's algorithm to implement an upgrade operation `upgrade`, from $\mathrm{VC}(n,b)$ to $\mathrm{VC}(n+x, \delta \cdot b)$ with $x \geq 0$ more nodes and a factor $\delta \geq 1$ more bandwidth. We use μ to denote the embeddings.

Kraken first pre-computes the *slotCount*-values for the entire substrate network, i.e., for each substrate node v (a server or switch). Subsequently, Kraken computes the new center-of-gravity CoG for VC which minimizes the reconfiguration costs in terms of the number of to be released, i.e., migrated CUs M (function `minMigs`) and embedding footprint F (function `footprint`), by iterating over all nodes in the substrate. Subsequently, the best found solution is embedded (function `computeEmbedding`).

1) Minimal Migrations: To compute the minimal number of migrations, function `minMig` proceeds as follows, see Algorithm 2: For each node v in the substrate (i.e., all servers and switches), it computes a list of CUs which have to be "released" (i.e., put in a pool of CUs which will be embedded somewhere else by the algorithm), to be able to realize the new center-of-gravity at node v.

Algorithm 2 `minMig`(substrate node v)

Output: set of CUs

```
1: M ← ∅
2: L ← computeConflictLinks(v)
3: sort L with decreasing distance from v
4: for all links ℓ ∈ L do
5:    while ℓ oversubscribed do
6:       let c be an arbitrary CU below ℓ
7:       M ← M ∪ {c}
8:    end while
9: end for
10: M ← M ∪ extraCUs(v)
11: return M
```

Algorithm 3 `footprint`(substrate node v, number of CUs to migrate m)

Output: cost value

```
1: done ← 0
2: for all children v' of v in the fat-tree do
3:    done ← done + slotCount(v')
4: end for
5: return ST(v) + height(v) · n + costsAbove(v, m − done)
```

`ComputeConflictLinks` computes the set of links L whose capacity would be oversubscribed if the center-of-gravity *cog* was on v and the bandwidth was increased to $b \cdot \delta$ under the current embedding μ of the existing CUs. Subsequently, we iteratively release CUs until a critical link $\ell \in L$ is no longer oversubscribed. This yields the first part of the set M of CUs which need to be migrated. The conflict resolution is ordered by distance to the center-of-gravity.

While releasing the CUs so far in M ensures that no link is oversubscribed, additional CUs may have to be moved to guarantee that the center-of-gravity is realized at the desired physical node: thus, `extraCUs` adds more CUs to the set M, such that the sum of the CUs which are currently hosted below v and the cardinality of M reach $n/2$. To make v the center-of-gravity of the virtual cluster, it is necessary and sufficient that at least $n/2$ CUs are below v.

2) Minimal Footprint: After determining the number of CUs that have to be migrated, we compute the embedding footprint. Interestingly, Kraken can compute the embedding cost of a desired center-of-gravity *without* determining an explicit embedding of the new virtual cluster, by utilizing the *slotCount*-values.

The function `footprint` is described in Algorithm 3. It takes a desired center-of-gravity v and a target number m of CUs which are to be migrated. Let us first observe that the footprint of a virtual cluster can be computed via the following case distinction: (1) If v is a core switch, all CUs are located below v, and hence the distance between v and the CUs is three. Thus, $F(\mathrm{VC}) = 3 \cdot \beta$, where β counts the number of CUs which are embedded *below* CoG(VC). (2) If v is an aggregation switch of a pod, the CUs of VC are either located on servers in the same pod, or on servers in different pods. Clearly, all servers in the same pod are

Algorithm 4 `costsAbove`(substrate node v, number of flexible CUs x)

Output: cost value

1: **if** $z = 0$ **then**
2: **return** 0
3: **end if**
4: **if** (v is a core switch or the uplink from v does not have $z \cdot \delta \cdot b$ spare bandwidth) **then**
5: **return** ∞
6: **end if**
7: $done \leftarrow 0$
8: **for all** for all siblings v'' of v' **do**
9: $done \leftarrow done + slotCount(v'') + D_{v''}$
10: **end for**
11: **return** $2 \cdot z + \texttt{costsAbove}(v', z - done)$

at distance two from v, and the servers in other pods are at distance four from v. We have $F(\mathrm{VC}) = 2 \cdot \beta + 4 \cdot \alpha^{(p)}$, where $\alpha^{(p)}$ is the number of CUs of VC which are embedded *above* CoG(VC), in a different pod. (3) In case v is embedded to a ToR switch, the embedding footprint is given by $F(\mathrm{VC}) = \beta + 3 \cdot \alpha^{(r)} + 5 \cdot \alpha^{(p)}$, where $\alpha^{(r)}$ is the number of CUs of VC which are embedded *above* CoG(VC), in a different rack. (4) The embedding footprint for a v on servers is given by $F(VC) = 2 \cdot \alpha^{(s)} + 4 \cdot \alpha^{(r)} + 6 \cdot \alpha^{(p)}$, where $\alpha^{(s)}$ is the number of CUs of VC which are embedded *above* CoG(VC), on a different server. In this case, CUs which are embedded *below* the CoG are omitted, as they have no bandwidth costs.

The function `footprint` first computes the number of CUs which can be placed on each of the sub-trees represented by the direct children of v. Since the center-of-gravity v is above its children by definition, the $slotCount(v)$-values of the children are accurate. Then, the embedding cost is computed recursively by the formula $\texttt{ST}(v) + height(v) \cdot n + \texttt{costsAbove}(v, z - done)$. The first cost term $\texttt{ST}(v)$ accounts for the static costs, i.e., the costs from CUs which are not scheduled for migration according to the minimal migrations. The second cost term $height(v) \cdot n$ depends on the depth of the center-of-gravity in the tree. The third term computes the additional costs from the CUs above v, if any, see the function `costsAbove` (Algorithm 4): we leverage the fact that the costs for placing CUs further away from a candidate center v increases by two for every layer in the fat-tree, regardless of the layer where v is located. Accordingly, given z flexible CUs, we add $2z$ to the costs and execute the function again with the parent node of v as the new v and $z - \sum_{v' \in V'} slotCount(v')$ as the new z, where V' is the set of siblings of v (i.e., children of the parent node of v excluding v). If v is the core switch, or the spare capacity on the uplink of v is less then $z \cdot \delta \cdot b$, v cannot be the center-of-gravity, and the upgrade request fails for this specific location of the CoG. If this is the case for all nodes v in the substrate, the upgrade request has to be rejected.

D. Downgrade Algorithm

Downgrade operations in Kraken never require any migrations. However, the center-of-gravity may change. Thus, the downgrade algorithm of Kraken proceeds similar to the upgrade algorithm, but without functions `minMig` and without the need to compute the $slotCount(v)$ values. The main difference regards how the values are actually used to compute the costs. While the original algorithm depends on *slotCount*-values and the current distribution, we set the current distribution to 0 and all *slotCount*-values to the distribution prior to the upgrade.

E. Formal Guarantees

Since the calculated cost and slotCount values are *exact*, we have derived the following result.

Theorem IV.1. *Kraken guarantees:*

1) Request Satisfiability: *As long as a feasible solution exists all upgrade and downgrade requests are satisfied.*
2) Minimal Reconfiguration: *The reconfiguration costs is always minimized. In particular, if a solution without migrations exists, it is used.*
3) Optimal Allocation: *Among all possible solutions with minimal reconfiguration costs, Kraken computes the one with the minimal embedding footprint.*
4) Complexity: *The time complexity of re-configuring (or embedding) a virtual cluster is bounded by $O(N \cdot n \cdot \Delta)$ in the worst-case, where N is the size of the substrate (number of servers), n is the virtual cluster size, and $\Delta = S + R + P$ is the number of servers in a single rack S (i.e., the degree of a ToR switch), plus the number of racks in a single pod R (i.e., the degree of an access switch), plus the number of pods P (i.e., the degree of a core switch).*

Note that Kraken can also be used to embed virtual clusters from scratch, and ensuring a minimal footprint. Thus, together with property *4)*, Kraken also outperforms state-of-the-art virtual cluster embedding algorithms which do not support any reconfigurations, e.g., [9], [21].

F. Alternative Migration Cost Models

For ease of exposition, we presented Kraken for a simple model where the objective of minimizing the number of migrations is prioritized over optimizing the embedding footprint. However, our algorithms can be extended to other migration cost models and trade-offs between migration and footprint costs, without sacrificing optimality. For instance, intra-pod migration costs could be modeled to be cheaper than inter-pod migrations, and migration costs could also depend on the available bandwidth along the migration path. Further, our algorithms support objectives describing arbitrary (weighted) linear combinations between the migration and footprint costs: e.g., if smaller resource footprints can be achieved with more migrations, they can also be computed.

V. Evaluation

We conduct extensive simulations to study the feasibility of online reservation upgrades at runtime. By default, we will assume the same settings and parameters as used in previous work [5]. However, given our more dynamic environment, we also introduce a model for elastic reconfiguration requests, and conduct a sensitivity analysis, studying the impact of different factors (such as magnitude of reconfiguration and system load) by using parameter sweeps.

A. Metrics

We consider the following two metrics:

Acceptance Ratio. Ideally, a system such as Kraken should be able to accept and satisfy as many requests as possible. For each request (either arrival of a new virtual cluster or a reconfiguration request), we distinguish whether or not the request was satisfied and, if satisfied, whether it was satisfied (1) *with* or (2) *without migrations*. Note that Kraken does not use "strategic access control" (e.g., to favor "small" requests to improve that acceptance ratio); in fact, Kraken never rejects a request if it can be satisfied.

Reconfiguration Costs. While our simulation does not capture many parameters that determine the actual cost of a migration we count the number of migrations; this is a natural metric given the uniform size of CUs of the virtual cluster. In particular, we will report on the *fraction* of migrated CUs relative to the virtual cluster size, which provides more insights than an absolute number.

B. Methodology & Runtime

Substrate. We model the datacenter as a three-level fat-tree. Overall, we have 16,000 servers distributed over $P = 10$ pods of $R = 40$ racks each; a rack contains $S = 40$ servers. By varying the connectivity and the bandwidth of the links between the switches, we change the over-subscription of the physical network. By default, we will assume that the access network is oversubscribed by a factor $\gamma_1 = 4$, while the core is not oversubscribed ($\gamma_2 = 1$). The available bandwidth is $B = 10$ Gbps.

Demand. New virtual cluster requests arrive according to a Poisson process with $\lambda = 0.36$. The lifetime of each virtual cluster is chosen according to an exponential distribution with average 3,600 s (one hour). By default, the size of a virtual cluster and the bandwidth are chosen from an exponential distribution with mean 49 and 2.5 Gbps respectively. The parameters are normalized to induce a system load of 0.75 on average. The size of the virtual cluster in numbers of CUs is chosen randomly from an exponential distribution, with an average of 49 CUs per cluster.

Elastic Model. To add dynamicity to the virtual cluster demands, we use six additional Poisson processes which continuously pick virtual clusters for upgrading and/or downgrading in a *multiplicative manner*. More precisely, the embedded clusters are continuously reconfigured by these six independent processes which randomly choose one of the existing clusters and perform a multiplicative update, i.e., either (1)+(2) upgrade or downgrade the bandwidth by a factor f_b (f_b corresponds to δ in our formal sections), (3)+(4) increase or decrease the cluster size by a factor f_n (f_n is the multiplicative version of the additive x in our formal sections), (5)+(6) *jointly* upgrade or downgrade the bandwidth *and* the cluster size by a factor f. By default, we assume that $f = f_b = f_n = 1.5$. With regards to reporting the results, we focus on the upgrades as these are the ones which trigger migrations.

To ensure the statistical significance, we run our simulations for 80k rounds which is roughly eighty times the duration (i.e., lifetime) of a virtual cluster. To avoid artifacts related to the initial empty substrate, we omit the first 10k requests.

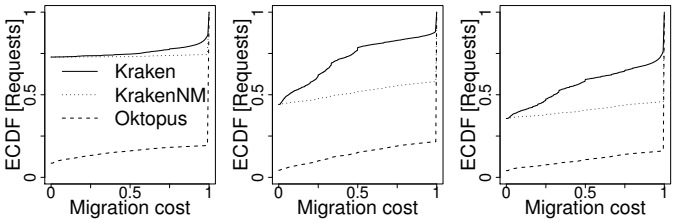

Fig. 3: Reconfiguration costs: KrakenNP vs. Kraken vs. Baseline (augmented Oktopus)—(*left:*) cluster size upgrade, (*middle:*) bandwidth upgrade, (*right:*) joint upgrade. The legend on the *left* is valid for all three plots.

Runtime. In this scenario, Kraken requires 86 ms on average to satisfy any given request (the 99^{th} percentile is 344 ms), when run on an Intel i3-2310M CPU @ 2.10GHz.

C. Baseline Comparison

Kraken features two main mechanisms for the efficient upgrade of a virtual cluster: (1) Kraken allows to upgrade an existing embedding by increasing the bandwidth between CUs *at their current locations*, as well as by the extending the cluster by the local addition of new CUs; (2) if a local extension is not sufficient to satisfy a request, Kraken also supports the re-embedding, i.e., migration of existing CUs.

In order to understand the contribution of each of these two features, we break down the analysis of Kraken into two steps: We first study a variant of Kraken, called KrakenNP, which does not perform fine-grained migrations. (*NM* stands *No (local) Migrations*.) That is, KrakenNP is equivalent to Kraken, but if a request cannot be satisfied with the given CU embedding, it resorts to embedding the virtual cluster with the new specification from scratch. Subsequently, we study the full-fledged Kraken system which can migrate CUs arbitrarily in order to satisfy requests (subject to the usual constraint that the number of migrations should be kept minimal). For a simple baseline comparison, we also re-implemented Oktopus [5]; we extended Oktopus so that requests can be satisfied by re-embedding.

To give a basic understanding of the number of migrations required to support elastic virtual clusters, Figure 3 plots the empirical cumulative distribution function (ECDF) of the migration cost for the three algorithms KrakenNP, Kraken and Oktopus, and the three operations: add CUs, upgrade bandwidth, and joint upgrade of CUs and bandwidth. Note that when a new embedding is performed to satisfy an upgrade request, the mechanism will guide the embedding process to a similar configuration. This means that when possible, the CUs will be assigned to the same old location, which, hence, will not be counted toward the migration cost. This explains why in some cases the migration cost of Oktopus and KrakenNP can also have values different from zero (no migrations) and one (all CUs are migrated).

We first discuss a scenario where only the bandwidth is upgraded. In Figure 3 (*middle*), we can observe that already KrakenNP is far superior to Oktopus as it can satisfy 45% of

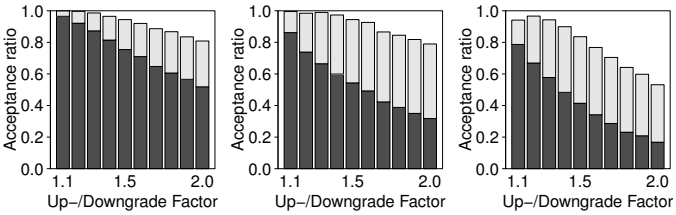

Fig. 4: Kraken acceptance ratios: without migration (*dark gray*), with migration (*light gray*)—(*left:*) cluster size upgrade, (*middle:*) bandwidth upgrade, (*right:*) joint upgrade.

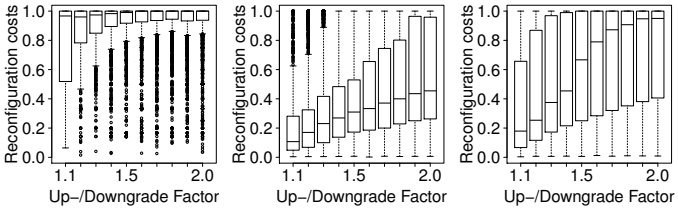

Fig. 5: Kraken reconfiguration costs for upgrades with migrations—(*left:*) cluster size upgrade, (*middle:*) bandwidth upgrade, (*right:*) joint upgrade.

the upgrade requests without migrations at all, while Oktopus has to migrate all CUs of a VC for 80% of the upgrade requests. In general, we find that Oktopus will likely find similar embeddings (with few migrations) if the upgrade request happens temporally close to the embedding time. However, later it becomes likely that virtual clusters will be embedded on a different sub-tree (or pod), resulting in many migrations. The performance of Kraken is very similar to the one of KrakenNP. However, the missing support of partial and coordinated migrations leads to $\approx 50\%$ cases where KrakenNP has to migrate all CUs, while Kraken can avoid migrating more then 50% of the CUs for nearly 80% of the requests.

The corresponding results for cluster size upgrades are shown in Figure 3 (*left*). While Oktopus can only embed about 10% of the upgrade requests without migrating any CUs, Kraken can upgrade 70% of the requests without migration. KrakenNP achieves a similar performance, and only for 10% of the requests, we can observe an improvement $\geq 5\%$ with Kraken in terms of reconfiguration costs.

Figure 3 (*right*) studies joint upgrades (bandwidth and cluster size). Here, the overall performance of Oktopus remains the same, and the performance of Kraken and KrakenNP becomes a mixture of the previous cases. While both variants of Kraken need no migrations for 35% of the requests, KrakenNP has to migrate all CUs for 40% of the requests, while Kraken can satisfy about 70% of all requests without migrating all CUs.

D. Sensitivity Study

Next, we conducted a sensitivity study of Kraken, in which we performed parameter sweeps for the up- and downgrade ratios f_b and f_n, the mean number of CUs per request, the bandwidth requirements per CU, the substrate load, and the access network over-subscription ratio. We will first study the effect of the upgrade ratios $f_b = f_n$ in greater detail, and subsequently, we report on our general observations for the other parameters.

Figure 4 shows the acceptance ratio for virtual cluster upgrades as bar plots. The dark gray area corresponds to upgrade requests that do not require migration. The light gray component of the bar corresponds to those requests that can be satisfied by Kraken but require migration. We again have three subplots corresponding to the three operations: adding CUs, upgrading bandwidth, and joint upgrades of CUs and bandwidth. The impact of the upgrade factor f is significant, opening a spectrum from "accepting almost all requests without migrations" (for factors close to one) to "no migration for only 50% of the cluster size upgrade requests". The impact of f on the bandwidth upgrades is even more articulated. As expected in the joint upgrade scenario, the two factors are amplified. Indeed, the problem is unfeasible for more than 40% of the requests if the upgrade factor is 2.

To better understand the difference between adding CUs and upgrading the bandwidth, Figure 5 zooms into the light-gray area and plots the distribution of the relative number of migrations, *given that the upgrade required at least one migration*. While in most cases it is sufficient to migrate less than half of the CUs for bandwidth upgrades, it is necessary to migrate more than 90% of the CUs, if any reconfigurations are necessary during a size upgrade. This can be explained by the different triggers of migrations for the two operations: In many situations, the CUs of a VC are collocated with each other. Adding CUs in this cases does not require reconfigurations, as long as there is sufficient spare bandwidth on the subtree, which currently hosts the VC. Contrary, even a small bandwidth upgrade can change the maximum number of CUs which can be collocated (e.g., a bandwidth upgrade from 2.4 Gbps to 2.6 Gbps changes maximum number of collocated CUs from 4 to 3), which will require a share of the CUs (in this case 25%) to be migrated. The only case in which adding CUs will actually trigger migrations, occurs when the subtree which currently hosts the VC is already highly filled, and the center has to be moved in order to meet the bandwidth guarantees. This can also happen during a bandwidth upgrade, but the first case occurs more often, and hence has a strong impact on the outcome shown in Figure 5. The joint upgrade case, shows the combined effects of the other two described upgrades.

We will now report on our observations for the other parameters: Varying any of the above parameters by 50% never caused the acceptance ratio to drop below 80%. Moreover, the acceptance ratio for CU as well as bandwidth upgrades are comparable to those of Figure 4. Joint upgrades are slightly more complex but the acceptance ratio is still above 80%. The largest difference we observed in the worst case acceptance ratio was 6%.

With regards to the reconfiguration costs we find that cluster size upgrades are typically more expensive. This is fully consistent with the observations above. It also points out that even local greedy search strategies for re-embedding CU size upgrades can be fairly successful.

In general, we see that most parameters only have a very small effect on the reconfiguration costs of bandwidth upgrades, and a small effect on the joint upgrade. On average

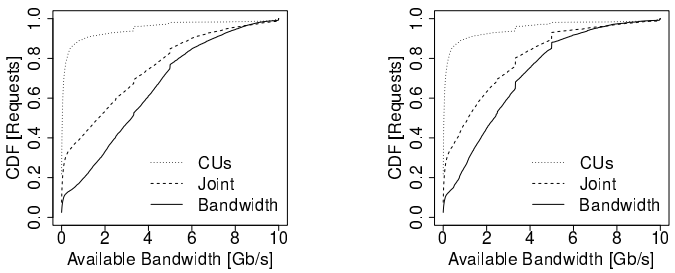

Fig. 6: CDF of the available bandwidth to migrate a compute unit for upgrades which require migrations. *Left:* avg. bandwidth; *Right:* min. bandwidth.

across all evaluated parameters, bandwidth upgrades need approximately one third reconfigurations per CU, while joint upgrades typically require two third reconfigurations per CU. This indicates that these operations benefit from the rigorous optimizations of Kraken.

E. Bandwidth for Migrations

While compute units can be small and light-weight, it may sometimes be desirable to migrate more state or entire VMs. Therefore, we investigate the bandwidth available during CU migrations. Figure 6 shows that for bandwidth upgrades, on average, approximately 3 Gbps can be guaranteed along the migration path of each CU on average; the minimum is around 2 Gbps. For joint upgrades, the values are 2 Gbps on average and 1 Gbps for the CU with the lowest available bandwidth. These values are encouraging, indicating that even large migrations are feasible in reasonable time. However, we also see that on the occasion where cluster size upgrades trigger migrations, the bandwidth can become critical: only 10% of the requests can guarantee more then 1Gbps of bandwidth for the migrations. In such settings, one may have to resort to a separate management network for migration.

VI. Discussion

This paper presented the Kraken system which allows to dynamically scale up and down the bandwidth and compute resources allocated to a cloud application at runtime. Thus, Kraken overcomes the weaknesses of existing solutions, in which resource reservations either cannot be changed [5], [11], [20], in which the entire resource schedule has to be computed at job submission time [7], or in which either only the bandwidth or the compute resources can be adapted, but not both [7], [18], [23].

We described algorithms to find a configurable and optimal tradeoff between embedding and reconfiguration costs, and complemented the formal guarantees by simulation.

While we have motivated our approach (and are currently implementing a prototype) for batch-processing applications such as MapReduce, we believe that our solution is of more general interest. It also complements nicely the recent work on *time malleable* systems like Amoeba [2] and Natjam [6] or scheduling frameworks such as Jokey [8]. Kraken can also be applied to systems such as Bazaar [12] that provide a job-centric interface and allow the provider to select the best combination of CUs and network resources. The ability of reallocating CUs and network resources at runtime can expand the range of scheduling opportunities.

Acknowledgments. Research supported by the German BMBF Software Campus grant 01IS12056, by the German-Israeli Foundation for Scientific Research and Development (GIF No I-1245-407.6/2014), and by the German Ministry for Education and Research (Berlin Big Data Center, BBDC).

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, 2008.
[2] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *ACM SOCC*, 2012.
[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, 2013.
[4] 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 *USENIX OSDI*, 2010.
[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, 2011.
[6] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *ACM SOCC*, 2013.
[7] D. Xie et al. The only constant is change: incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*, 2012.
[8] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *7th ACM EuroSys*, 2012.
[9] C. Fuerst, M. Pacut, P. Costa, and S. Schmid. How hard can it be? understanding the complexity of replica aware virtual cluster embeddings. In *Proc. 23rd IEEE International Conference on Network Protocols (ICNP)*, 2015.
[10] 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 *ACM SIGCOMM*, 2009.
[11] 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 *ACM CoNEXT*, 2010.
[12] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SOCC*, 2012.
[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune in action: Mitigating skew in mapreduce applications. *VLDB Endow.*, 5(12), 2012.
[14] L. Popa et al. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.
[15] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014. USENIX Association.
[16] M. Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.
[17] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.
[18] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, 2010.
[19] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM*, pages 351–362, 2013.
[20] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *3rd Conference on I/O Virtualization (WIOV)*, 2011.
[21] M. Rost, C. Fuerst, and S. Schmid. Beyond the stars: Revisiting virtual cluster embeddings. In *ACM SIGCOMM CCR*, 2015.
[22] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *USENIX HotCloud*, 2010.
[23] K. C. Webb, A. Roy, K. Yocum, and A. C. Snoeren. Blender: Upgrading Tenant-based Data Center Networking. In *ANCS*, 2014.
[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, 2008.
[25] Measuring EC2 system performance. http://goo.gl/V5zhEd.