Resource Management in Multi-* Clusters: Cloud Provisioning

Mosharaf Chowdhury
mosharaf@cs.berkeley.edu

April 30, 2010

Abstract

Cloud computing – running large-scale computation- and data-intensive services on inexpensive on-the-fly clusters – has become increasingly popular in recent years due to the advent of MapReduce and similar large-scale data parallel systems. Cloud provisioning, i.e., allocating resources for cluster requests, is the first step toward instantiating such clusters.

In this report, we formally define the *cloud provisioning problem (CPP)* and show it to be $\mathcal{NP}$-hard. We present a flow network-based model of CPP and describe a generalized framework for optimally solving variants of CPP (e.g., CPP with load balancing (LBCPP), fault tolerance (K-FTCPP) etc.) using linear and non-linear mixed integer programs. Since mixed integer programs are generally $\mathcal{NP}$-hard, we also sketch a linear programming relaxation and randomized rounding-based approximation algorithm for 1-FTCPP (a specialized/simpler variant of K-FTCPP) and propose a rounding-based heuristic for the problem.

1 Introduction

With the advent of MapReduce [6], Dryad [10] and similar frameworks as well as of cloud services like Amazon's EC2[1], cluster computing has become mainstream in recent years. Consequently, the complexity of resource allocation and scheduling in such multi-tenant, multi-framework, and multi-user environments have increased considerably.

In the beginning of the cloud-age, resource management problem was all about scheduling online homogeneous jobs (e.g., MapReduce jobs) with heterogeneous requirements from different users in privately owned clusters [3, 9, 12]. Next, cloud providers started to offer services to create on-demand clusters (e.g., EC2). This added a new dimension where cloud providers first need to allocate resources from a large pool, and then their clients get to implement their own scheduling mechanisms in the leased resources (sometimes cloud providers offer the scheduling service as well). Finally, in order to support heterogeneous frameworks (e.g., MapReduce and MPI) simultaneously, research on cluster computing operating systems (OSes) (e.g., Nexus [8]) is gaining momentum.

Resource management problems in multi-* cluster environments can now be classified into three large categories. Putting everything together, the resource management process in such environments roughly consists of:

- Cloud providers provisioning/delivering raw clusters based on resource requirements of their customers;
- Customers running cluster OSes to manage those resources to manage and schedule jobs from multiple frameworks; and

[1] http://aws.amazon.com/ec2/

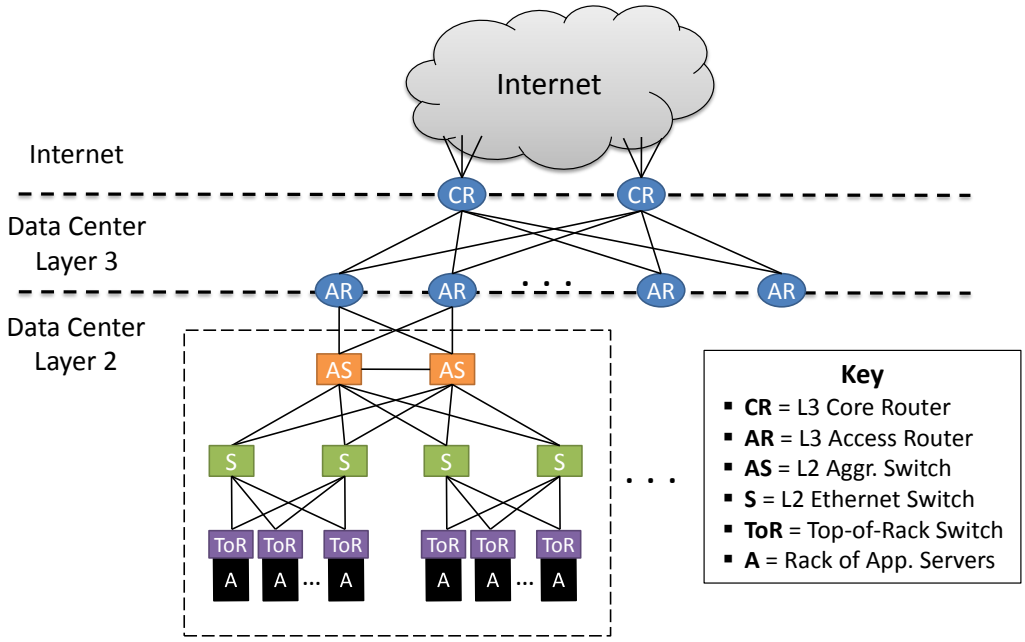

Figure 1: A conventional network architecture for data centers. Adapted from figures by Cisco [1] and Greenberg et al. [7].

- Frameworks scheduling tasks[2] (with or without assistance from the cluster OS) to get the job done.

The overarching goal of a comprehensive solution to the resource management problem in multi-* clusters may include: creating a theoretical model taking all three aforementioned stages into account, analyzing their individual and combined behavior, formulating mechanisms that optimize contrasting goals of the involved parties in different stages, and finally, providing theoretical bounds on the individual and combined performance of the proposed mechanisms.

Existing work in this space ranges from fair and efficient schedulers for scheduling tasks [9, 12], heuristics for decentralized scheduling to handle multiple frameworks [8], to resource allocation mechanisms for raw clusters. However, most mechanisms are straightforward heuristics based on empirical evidences without formal definition or study of the complexities of the problem or the algorithms.

In this report, we focus on the *Cloud Provisioning Problem (CPP)*, i.e., the online resource allocation problem faced by the cloud providers. In Section 2, we present an overview of data center architecture and cloud provisioning landscape. We formally define CPP in Section 3. Section 4 presents a flow network-based formulation of CPP followed by linear and non-linear mixed integer programming representations of different variants of CPP. We attempt to present an approximation algorithm for 1-FTCPP, which we believe to be one of the less complex variants of CPP, in Section 5. We discuss possible extensions, future work, and status of the simulation evaluation in Section 6 and conclude in Section 7.

2 Background

In this section, we briefly explain the dominant design pattern in current data center networking architecture [1] along with the consequences of such a design [7]. In addition, we also discuss the prevalent model of cloud provisioning.

[2] A job consists of multiple smaller tasks.

2.1 Data Center Network Architecture

The network architecture inside a data center is basically a hierarchy reaching from a layer of servers in racks at the bottom to a layer of core routers at the top (Figure 1). There are typically 20 to 40 servers in each rack, and each of them is connected to a Top-of-Rack (ToR) switch with a 1 Gbps link. ToRs connect to two aggregation switches for redundancy. All switches below each pair of access routers form a single layer-2 domain, typically connecting several thousand servers. Aggregation switches further aggregates to access routers. At the top of the hierarchy, core routers carry traffic between access routers and manage traffic between the Internet and the data center. The consequences of such a design are as follows [7]:

As traffic moves up through the layers of switches and routers, the over-subscription ratio increases rapidly. For example, servers typically have 1:1 over-subscription to other servers in the same rack (i.e., 1Gbps communication speed), whereas, up-links from ToRs are typically 1:5 to 1:20 oversubscribed (i.e., 1 to 4 Gbps of up-link for 20 servers), and paths through the highest layer of the tree can be 1:240 over-subscribed. This large over-subscription factor translates into high communication costs between racks and layer-2 domains.

Moreover, spreading a service outside a single layer-2 domain often requires reconfiguring IP addresses and VLAN trunks. To avoid this, spare capacity throughout the data center often has to be reserved for individual services (and not shared), so that each service can scale out to nearby servers to respond to dynamic demand spikes or failures without disrupting other services (due to their eviction or migration).

Finally, above the ToR, the basic resilience model is 1:1, i.e., the network is provisioned such that if an aggregation switch or access router fails, there must be sufficient remaining idle capacity on a counterpart device to carry the load.

2.2 Cloud Provisioning

Any application needs a model of computation, a model of storage, and a model of communication [3]. Cloud providers virtualize each of these resources and statistically multiplex them to achieve elasticity and to present the illusion of unlimited capacity.

Offerings can widely vary from one cloud provider to another. Amazon EC2 is at one end of the spectrum that offers different instance types with varying computation, storage, and communication capabilities for different prices. Each EC2 instance looks much like a physical machine where users can control nearly the entire software stack. However, such user control makes it inherently difficult for Amazon to offer automatic scalability and failover, because the semantics associated with each instance are highly application-dependent.

At the opposite extreme are application domain specific platforms such as Google AppEngine[3]. AppEngine is targeted exclusively at traditional web applications and enforces an application structure of clean separation between a stateless computation tier and a stateful storage tier. AppEngines impressive scalability and high-availability characteristics rely on these constraints.

3 Cloud Provisioning Problem (CPP)

We focus on cloud providers who provide the highest level of flexibility to their customers (e.g., Amazon EC2), i.e., cloud providers that provide almost raw machines/resources based on customer requests. This is necessary because without complete flexibility our overarching goal of three-stage fine-tuned resource management will not be feasible. We also assume that such a provider implements a conventional hierarchical data center network layout (Section 2) to deploy servers and to interconnect them across racks throughout the data center.

[3] `http://code.google.com/appengine/`

Definition 3.1 (Cloud Provider Network (CPN)) *A cloud provider network (CPN) is given by a weighted undirected graph $G^P = (N^P, L^P, A^P)$, where N^P is the set of server nodes and L^P is the set of links interconnecting the servers. Each node or link $e \in (N^P \cup L^P)$ is associated with a set of capacity attributes $A^P(e) = \{A_1^P(e), \ldots, A_D^P(e)\}$, where D is the total number of attributes.*

Examples of node or server attributes include CPU ($A_c^P(.)$), memory ($A_m^P(.)$), and local disk ($A_d^P(.)$) capacities. Examples of link attributes include bandwidth ($A_c^P(.)$) and communication cost ($A_t^P(.)$) (e.g., links within the rack have the lowest costs, whereas links connecting aggregation switches to access routers have very high costs).

Requests for clusters (i.e., a collection of data center resources) come from customers in an online fashion one after another. Normally, cloud providers have predefined set of instances that customers can request for (e.g., EC2 accepts requests for server nodes specific for memory-intensive and CPU-intensive tasks as well as generic servers). However, we can generally define cluster requests in a way similar to CPNs. In addition to defined requirements, a customer can optionally provide a set of hints forecasting its peak requirements.

Definition 3.2 (Cluster) *A cluster is defined by a weighted undirected graph $G^R = (N^R, L^R, A^R)$ similar to a CPN. N^R and L^R are the sets of requested servers and their interconnections. Each node or link $e \in (N^R \cup L^R)$ is associated with a set of requirement attributes $A^R(e) = \{A_1^R(e), \ldots, A_D^R(e)\}$.*

Definition 3.3 (Cluster Request) *A cluster request consists of a cluster definition G^R, its arrival time a^R, and a set of performance characteristics hints H^R.*

When a cluster request arrives, the cloud provider has to determine whether to accept the request or not. If the request is accepted, the cloud provider then determines a suitable assignment for the request and allocates resources on server racks by means of creating virtual machines selected by that assignment. The allocated resources are released once the VN expires. However, if required, virtual machines can be migrated [11] in real-time without disrupting running services (mass migration can create significant traffic fluctuation though). We define cluster assignment as a mapping function from the cluster requested to CPN.

Definition 3.4 (Cluster Assignment (CAT)) *An assignment (CAT) of a cluster $G^R = (N^R, L^R, A^R)$ onto a CPN, $G^P = (N^P, L^P, A^P)$ is defined as a mapping of G^R to a subset of G^P such that each requested virtual instance is mapped onto exactly one physical server and each virtual interconnection can be mapped onto a loop-free path in CPN:*

$$\mathcal{M} : G^V \rightarrow (N^P, \mathcal{P}^P)$$

where $\mathcal{P}$ denotes the subset of all loop-free paths in G^P. $\mathcal{M}$ is called a valid CAT if all requirements A^V of the cluster request G^V have been satisfied and for each $l^V = (s^V, t^V) \in L^V$ there exists a path $p(s^P, t^P) \in \mathcal{P}^P$ with $M(s^V) = s^P$ and $M(t^V) = t^P$. A CAT can naturally be decomposed into node and link assignment as follows:

$$\textit{Node Assignment: } \mathcal{M}_N : N^V \rightarrow N^P$$
$$\textit{Link Assignment: } \mathcal{M}_L : L^V \rightarrow \mathcal{P}^P$$

Definition 3.5 (Residual Capacities) *Given a CPN G^P, a cluster request G^V, and a CAT ($\mathcal{M} : G^V \rightarrow G^P$), we get the residual capacities of nodes and links (denoted by A^{P_R}) of G^P by subtracting from them the requirement attributes of each requested node and link of G^V that are mapped onto them.*

Revenue from a single CAT is collected, when customers leave freeing the servers, using a flat-rate based on the resources that had been in use.

Definition 3.6 (Revenue ($\mathbb{R}$) and Cost ($\mathbb{C}$)) *Revenue from a single CAT, $\mathcal{M}$ that was active for time period t^R is defined as the sum of cost of total server resources in use multiplied by t^R.*

$$\mathbb{R}(\mathcal{M}) = t_R \times \sum_{\substack{A_z^R \in A^R(n^R) \\ n^R \in N^R}} \mathbb{C}(A_z^R) \tag{1}$$

where $\mathbb{C}(.)$ is the cost function for corresponding resource.

Note that, we have left server interconnection costs out of the revenue function in compliance with prevalent revenue models. However, if required, they can also appear as an additive term in $\mathbb{R}$.

Definition 3.7 (Cloud Provisioning Problem (CPP)) *Given a CPN, G^P and online cluster requests, G_i^R, the cloud provisioning problem (CPP) finds valid CATs, $\mathcal{M}_i$ for each G_i^R such that revenue $\mathbb{R}$ of the cloud provider is maximized in the long-run.*

While finding a valid CAT, CPP might take several strategies based on cloud provider policies resulting in different variations of CPP. Examples of such strategies/variations include:

- *Cost minimization strategy* (CMCPP): Putting as many virtual servers in the same rack as possible to reduce communication costs (since link costs are generally not charged for);
- *Load balancing strategy* (LBCPP): Distributing load from different cluster requests across the data center instead of filling a rack before moving onto the next to better handle demand spikes (because that will result in migration in the future, eventually incurring bandwidth and communication cost);
- *Fault tolerance strategy* (FTCPP): Distributing servers from the same request across physical machines, even across racks, instead of putting them close together for fault tolerance.

At this point, we are ready to show that CPP is very hard to solve; in fact, we prove it to be $\mathcal{NP}$-hard. We do this by reducing a variant of the bin-packing problem [5] to CPP in polynomial time.

Claim 3.1 *CPP is $\mathcal{NP}$-hard.*

Proof Sketch We can consider each node $n^P \in N^P$ as a bin and each requested virtual server $n^R \in N^R$ as an item to be packed into bins. In our case, each bin has D-dimensional resources (i.e., each server has D-attributes), and we have a fixed-number of bins (i.e., total number of servers in the data center is fixed). Servers or bins can have variable capacities, i.e., they do not have to have capacity of exactly 1. Moreover, requests can arrive at any moment and leave as well, and they can be moved around (i.e., migrated) for better assignment; in short, we are handling a *fully dynamic* bin-packing problem. Now, we can easily show:

D-dimensional fully dynamic fixed-number bin packing problem with variable bin sizes $\leq_P$ CPP

If we have a black-box that can solve CPP, we can solve the *D-dimensional fully dynamic fixed-number bin packing problem with variable bin sizes* by appropriately scaling down resource attributes of G^P and G^R in CPP within $[0..1]$. ■

Note We can reduce the vanilla bin-packing problem to CPP in polynomial time to show CPP to be $\mathcal{NP}$-hard as well. But we chose the very specific variant due to its uncanny similarity.

4 Flow-based Cloud Provisioning

We introduce a flow-based framework for cloud provisioning in this section. We represent cloud provisioning as a graph and encode both the structure of the CPN and cluster requests from customers in that graph.

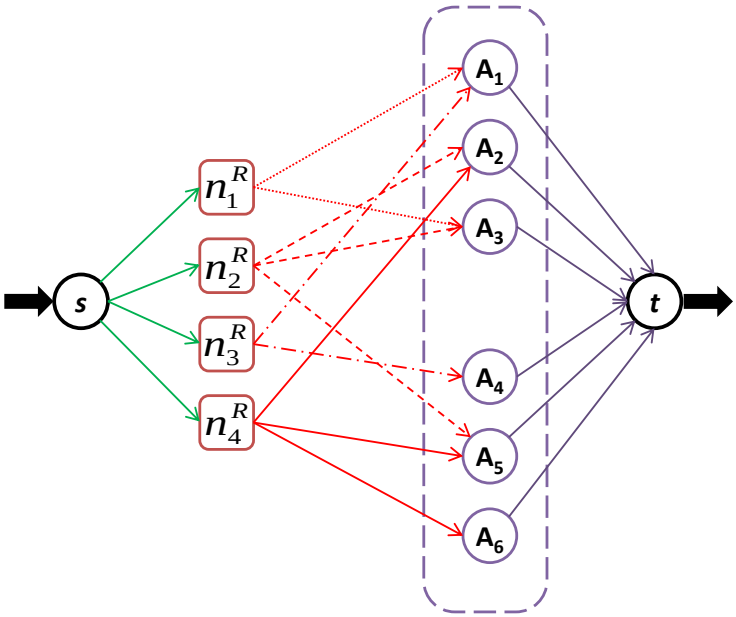

Figure 2: Flow network representation of CPP. A_i are the servers in CPN, n_j^R are the requested servers, and t is the sink node. Edges from n_j^R to $\{A_k\}$ represent that those A_k have capacity attributes to satisfy requirement attributes of n_j^R.

By assigning appropriate weights and capacities to the edges in this graph, we formulate a declarative description of our provisioning scheme. Similar approaches exist for job scheduling in clusters [9] and for virtual network embedding [4].

The primary intuition that allows a graph-based declarative description of our problem is that there is a quantifiable *opportunity cost*[4] associated with decisions made by different provisioning strategies. For example, there is a communication cost of assigning cluster nodes on particular servers or racks, and there is also a cost in wasted bandwidth in migrating already assigned nodes to different servers or racks. If we can approximate these costs in the same units as revenue using some conversion functions (e.g., in monetary terms), we can write down a mathematical program to try to minimize the total cost of an assignment. Since the revenue from a provisioning remains the same regardless of the exact details of that provisioning (because revenue function $\mathbb{R}$ uses a flat-rate), by minimizing the cost of each assignment we can effectively maximize the overall revenue.

4.1 Representing Cloud Provisioning as a Flow Network

In order to create the flow network G^F, we select the server nodes in N^P (skipping the internal nodes, e.g., switches, routers etc.) by adding nodes to it based on the requirement attributes of the nodes in the cluster request, G^R. Since each $n^R \in N^R$ has an associated set of attributes, we can select a set of candidate servers for each requested node ($\left|N^R\right|$ in total) in CPN such that each candidate physical server has more residual capacities than requested for each attribute. We denote such a set by $\Omega(n^R) \subseteq N^{P_S}$, where $N^{P_S} (\subset N^P)$ is the set of physical servers in CPN.

[4]We are using the term "opportunity cost" rather loosely here to refer to the fact that for any provisioning decision we make, we are probably giving up some revenue in return in the near future. For a brief overview of the concept of opportunity cost, please refer to: `http://en.wikipedia.org/wiki/Opportunity_cost`

$$\Omega(n^R) = \left\{ n^P \in N^P \mid \forall_{A_z^R \in A^R(n^R)} A_z^R(n^R) \leq A_z^{P_R}(n^P) \right\}$$

For each $n^R \in N^R$, we connect n^R with all physical nodes belonging to $\Omega(n^R)$ using edges of infinite capacity. We also connect a source node s to each requested node and each physical server node to a sink node t with unit capacity edges.

Finally, we combine everything to create the directed flow network $G^F = (N^F, L^F)$, where

$$\begin{aligned} N^F &= N^{P_S} \cup N^R \cup \{s, t\} \\ L^F &= \{(n^R, n^P) \mid n^R \in N^R, n^P \in \Omega(n^R)\} \cup \\ &\quad \{(s, n^R) \mid n^R \in N^R\} \cup \{(n^P, t) \mid n^P \in N^{P_S}\} \end{aligned}$$

4.2 Mathematical Programming Formulation

We can now express the flow network representation of CPP as a mathematical program such that a solution to it provides an assignment for a given cluster request. By putting in/out demand constraints on the source and the sink nodes, it can be ensured that exactly the required number of servers are selected; and by introducing restrictions on the edges connecting nodes in N^R and N^{P_S}, each node $n^R \in N^R$ can be forced to choose only one edge to connect itself to a single physical server in $\Omega(n^R)$.

Program 4.1 (Mathematical Programming Formulation of CPP)
Constants

- c_{uv}: Capacity of the CPN edge (u, v). Unless otherwise specified, $c_{uv} = 1,\ \forall (u, v) \in L^F$.
- l_{uv}: Cost of communication between nodes $u, v \in N^P$

Variables

- f_{uv}: Flow variable denoting the total amount of flow from u to v on the CPN edge (u, v).
- x_{uv}: Binary indicator variable for f_{uv}.

Objective Function

$$\text{minimize} \quad \mathbb{O}(f, x) \tag{2}$$

Constraints

$$f_{uv} \leq c_{uv} * x_{uv} \tag{3}$$

$$\sum_{w \in N^F} f_{uw} = \sum_{w \in N^F} f_{wu},\ \forall u \notin \{s, t\} \tag{4}$$

$$\sum_{w \in N^R} f_{sw} = \sum_{w \in N^{P_S}} f_{wt} = |N^R| \tag{5}$$

$$A_k^{P_R}(v) \geq \sum_{\substack{u \in N^R \\ (u,v) \in L^F}} A_k^R(u) * x_{uv},\ \forall v \in N^{P_S}\ \vee\ 1 \leq k \leq |D| \tag{6}$$

$$\sum_{w \in \Omega(m)} x_{mw} = 1,\ \forall m \in N^R \tag{7}$$

$$f_{uv} \geq 0 \tag{8}$$

$$x_{uv} \in \{0, 1\} \tag{9}$$

Remarks

- The objective function (2) tries to minimize the objective cost function. Different provisioning strategies can be implemented by using appropriate objective functions.
- Constraint set (3) enforces capacity constraints of the edges of the flow network. It also sets $x_{uv} = 1$ whenever $f_{uv} > 0$.
- Constraint sets (4) and (5) refer to the flow conservation and demand satisfaction conditions, which denote that the net flow to and out of a node is zero, except for the source node s and the sink node t.
- Constraint set (6) ensures that all the capacity attributes of the CPN nodes are respected.
- Constraint set (7) ensure that x_{uv} is set whenever there is any flow in either direction of the substrate edge (u, v).
- Finally, constraint sets (8) and (9) denote the real and binary domain constraints on the variables f_{uv} and x_{uv}, respectively.

4.3 Mathematical Programs for CPP Variants

Now that we have a framework to express CPP as a mathematical program, we can plug in different objective functions for $\mathbb{O}(f, x)$ in (2) and solve them to implement different strategies for cloud provisioning. We consider three such variants and discuss them in further details in the following.

4.3.1 CMCPP

The objective of the CMCPP strategy is to minimize the all-pair communication cost for the cluster request under consideration, which essentially means minimizing the sum of $l_{\mathcal{M}_N(u)\mathcal{M}_N(v)}$ for all $u, v \in N^R$.

$$\mathbb{O}_{CMCPP}(f, x) = \sum_{\substack{u,v \in N^R \\ u',v' \in N^{P_S}}} x_{uu'} * l_{u'v'} * x_{vv'} \tag{10}$$

Note Finding optimal $\mathbb{O}_{CMCPP}(f, x)$ requires solving a quadratic mixed integer program.

4.3.2 LBCPP

LBCPP tries to spread out cluster allocation over servers with more remaining capacities so that it will be less likely to result in mass migrations due to sudden spikes in demand in the future.

$$\mathbb{O}_{LBCPP}(f, x) = \sum_{\substack{u \in N^R \\ u' \in N^{P_S}}} \frac{x_{uu'}}{\sum_{1 \le k \le D} A_k^{P_R}(u')} \tag{11}$$

4.3.3 K-FTCPP

While provisioning a cluster, the cloud provider cannot guess what type of jobs the cluster will run or how it will be organized logically. Consequently, it is not possible to make the cluster completely tolerant to

physical server faults. However, by enforcing a limit on the number of requested servers to be allocated on a single physical server (or may be on a single rack), a cloud provider can limit the consequence of a failure[5].

By adding the following constraint to Program 4.1 we can ensure that at most K number of requested servers from the same request will be allocated on a single physical server:

$$\sum_{m \in N^R} x_{mw} \leq K, \ \forall w \in N^{P_S} \tag{12}$$

The value of K can be pre-defined by the provider, or it can allow the customers to set their desired level.

A softer way (i.e., where requests will not be refused because there was no way of satisfying (12)) would be to avoid constraint (12) and instead having an objective function with a very large penulty for having more than K requested servers on the same physical server.

$$\mathbb{O}_{K-FTCPP}(f, x) = \Big(\sum_{m \in N^R} x_{mw} - K \Big) * \texttt{BIG\_PENULTY} \tag{13}$$

Note Finding optimal $\mathbb{O}_{LBCPP}(f, x)$ and $\mathbb{O}_{K-FTCPP}(f, x)$ require solving linear mixed integer programs.

4.3.4 Combining Different Provisioning Strategies

While separately implementing some of these strategies without using a mathematical program might not be too hard, it becomes increasingly harder when multiple strategies must be used together. By using a mathematical programming framework, we can use various combinations of strategies without much trouble. For example, in order to enforce all three aforementioned strategies together we just need to add their objective functions together.

5 Approximation Algorithms for Cloud Provisioning

Unfortunately, solving quadratic or linear mixed integer programs is known to be $\mathcal{NP}$-hard. As a result, mathematical programs presented in Section 4 are not practically usable unless the problem size is really small. Alternatives include approximation algorithms and heuristics that will give accurate enough results in polynomial time. In this section, we try to create an approximation algorithm for 1-FTCPP using linear programming relaxation and randomized rounding (which is much simpler version of K-FTCPP).

We do not consider CMCPP further in this report due to its higher complexity being a quadratic mixed integer program. However, CMCPP can be approached using techniques for non-linear mixed integer programs (e.g., vector relaxation, semidefinite programming etc.).

5.1 Approximation Algorithm for 1-FTCPP

1-FTCPP is a special case of the K-FTCPP problem where each requested server must be allocated into different physical servers. In this case, constraint set (12) can be rewritten as the following:

$$\sum_{m \in N^R} x_{mw} \leq 1, \ \forall w \in N^{P_S} \tag{14}$$

We approach toward finding an approximation algorithm by relaxing the corresponding integer program and then rounding the relaxed solution to find a feasible solution.

[5]Most cluster frameworks are robust to failure to some extent. By limiting the total number of failures, cloud providers can assist automatic recovery mechanisms in those frameworks.

5.1.1 Linear Programming Relaxation of 1-FTCPP

By replacing constraint (9) on x_{uv} in Program 4.1 with the following we can convert it to a linear program.

$$x_{uv} \in [0, 1] \tag{15}$$

Note The solution to the relaxed problem will result in cost less than or equal to that of the mixed integer version, if a solution exists.

Note The relaxed version is also useful for providing bounds to find optimum solutions using standard branch and bound techniques.

5.1.2 Randomized Rounding

If we can round up and down values of x_{uv} without violating any of the constraints of Program 4.1, we will get a feasible solution. At this point, we have to figure out a rounding strategy and prove that it will indeed give a valid solution to 1-FTCPP.

If we scale the values of x_{mw} for each $w \in N^{P_S}$ in (14), such that $\sum_{m \in N^R} x_{mw} = 1$, we can use the scaled x_{mw} values to select $m \in N^R$ with probability x_{mw} for all $w \in N^{P_S}$. This will directly ensure that constraints (14) have not been violated. At the same time, this will result in some m being assigned to multiple ws and some not being assigned at all. Now for each $m \in N^R$ we have

$$\Pr[m \text{ has not been provisioned}] = \prod_{w \in \Omega(m)} (1 - x_{mw}) \leq (1 - \frac{1}{|\Omega(m)|})^{|\Omega(m)|}$$

where the last inequality follows from the fact that $\prod_{w \in \Omega(m)}(1 - x_{mw})$ is maximized when $\sum x_{mw} = 1$ (ensured by constraint (7)) and are evenly distributed. Since $(1 - \frac{1}{|\Omega(m)|})^{|\Omega(m)|} \leq \frac{1}{e}$, where $e \approx 2.718$ is the base of the natural logarithm, we conclude that

$$\Pr[m \text{ has not been provisioned}] \leq \frac{1}{e} \approx 0.268$$

The probability that any requested server $m \in N^R$ has been assigned is fairly high, but this is not good enough: there are $|N^R|$ requested servers and even though each one of them has a good chance of being provisioned, we cannot expect all of them to be provisioned simultaneously. We want each $m \in N^R$ to be provisioned with high probability.

However, unlike liner programming relaxation and randomized rounding-based solution to the set cover problem, we cannot perform the rounding process multiple times and then combine all the solutions together to get a higher probability in this case.

5.1.3 Aftermath: Rounding-based Heuristics

Unfortunately, we have reached a point where cannot proceed any further with randomized rounding. One straightforward polynomial time heuristic extracted from the dead end we have hit can be as follows:

1. Do a randomized rounding as before and identify set of unprovisioned nodes $M' \subset N^R$. If M' is an empty set, we are done.
2. Arbitrarily choose an $m' \in M'$ and assign it to $w' \in \Omega(m')$ with probability $x_{m'w'}$.
3. Remove m' and w' from further computation ($N^R = N^R - \{m'\}$ and $N^{P_S} = N^{P_S} - \{w'\}$), recalculate probabilities, and continue.

At each step, we are decreasing the number of unassigned requested servers by one, if the newly assigned server was not the only option for the requested server it has been taken away from.

6 Discussion and Future Work

We discuss a potpourri of topics regarding cloud provisioning, proposed models/frameworks and their extensions, and issues regarding their evaluation etc. in this section.

6.1 Extensions to the Flow Network

In Section 4 we discussed a flow network, where we connected request servers directly to the physical servers ignoring the complete physical hierarchy inside a data center. An obvious extension to this model, would be to include the left out portion. By doing so, we can formulate mathematical programs that can achieve more precise control over where to provision requested servers by adding more constraints or by appropriately setting the objective functions.

6.2 Cloud Provisioning with Look-ahead

CPP being an online problem, throughout the report, we have made the assumption that each cluster request will come one at a time and then some mathematical program has to be solved (or may be some heuristics will be used) to provision or to reject that request right away. While we cannot change the basic online nature of the problem, we can introduce a time window through which all the requests will be accumulated, and later all of them will be solved simultaneously using Program 4.1. The natural intuition here is that the more information the program has, the better provisioning decision it can make.

6.3 Heuristics

In addition to the rounding-based heuristic proposed in Section 5, there can be many simpler heuristics for CPP and its variants – most notably the plethora of heuristics proposed for the original bin packing problem and its variants (e.g., first fit, first fit decreasing, best fit, clique-graph conflicts etc.) [5]. Many of them can be applied to CPP or some variant of CPP with little modification. We defer measuring the effectiveness of such heuristics using simulation to the future.

6.4 Notes on Evaluation

We started working on modifying an open source virtual network embedding simulator [2] to fit into our model so that we can compare some naive greedy algorithms against mathematical program-based optimal solutions and different heuristics. Unfortunately we faced several road blocks - the biggest one being the lack of public data on the actual characteristics of problem input.

While there are some documentation from EC2 about the nature of cloud providers, similar information on cluster requests can vary widely. Specifically, we did not have any real data on the arrival rate, duration time, size, or typical composition of cluster requests. We could generally guess that the dynamic part of the problem might have a Poisson arrival rate and requests might have Exponential duration, but the exact parameters were still unknown. Moreover, size and composition of requests can also vary widely.

We leave running simulation experiments to verify our intuitions as future work depending on the completion of porting the simulator and getting sample traces to generate inputs with real-world characteristics.

7 Conclusions

In this report, we have formally defined the *cloud provisioning problem (CPP)* and showed it to be $\mathcal{NP}$-hard. We presented a flow network-based model of CPP and described a generalized framework for optimally

solving variants of CPP using linear (LBCPP, K-FTCPP) and non-linear (CMCPP) mixed integer programs. Since mixed integer programs are generally $\mathcal{NP}$-hard, we also attempted a linear programming relaxation and randomized rounding-based approximation algorithm for 1-FTCPP, a specialized/simpler variant of K-FTCPP. Unfortunately, this turned out to be highly non-trivial and we ended up with a randomized rounding-based heuristic. We look forward to comparing this heuristic with its greedy counterpart as well as optimal solutions once we have access to real traces and have a running discrete event simulator for this purpose.

References

[1] Data center: Load balancing data center services. Cisco Press, 2004.

[2] ViNEYard Simulator. `http://www.mosharaf.com/ViNE-Yard.tar.gz`, 2009.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[4] N. M. Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *IEEE INFOCOM*, pages 783–791, 2009.

[5] Edward G. Coffman, Jr. Janos, Csirik David, S. Johnson, and Gerhard J. Woeginger. An introduction to bin packing, 2004.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62, 2009.

[8] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Scott Shenker, and Ion Stoica. Nexus: A common substrate for cluster computing. Technical Report UCB/EECS-2009-158, EECS Department, University of California, Berkeley, Nov 2009.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[10] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[11] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *USENIX NSDI*, pages 229–242, 2007.

[12] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.