Anchor: A Stable Matching Framework for Managing Cloud Resources

Hong Xu, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—Cloud computing infrastructures need an effective way to manage server resources so that performance can be improved and costs can be reduced. Current solutions are proprietary and not customizable. In this paper, we present *Anchor*, a new resource management architecture that uses the stable matching framework to decouple policies from mechanisms when mapping virtual machines to physical servers. In *Anchor*, cloud clients and operators are able to express a wide variety of distinct resource management policies as they deem fit, and these policies are expressed as preferences in a stable matching framework. The highlight of *Anchor* is a new many-to-one stable matching theory that efficiently matches multiple VMs with heterogeneous resource needs to their servers, using a multi-stage deferred acceptance algorithm to resolve conflicts of interest. Our theoretical analysis shows the convergence and optimality of the algorithm. Our experiments with a prototype *Anchor* implementation and large-scale simulations demonstrate that the architecture is general enough to realize a diverse set of policy objectives, while providing superior performance and practicality.**

I. Introduction

Modern data centers heavily rely on virtualization [7] to flexibly multiplex different applications onto physical servers, in order to efficiently utilize their resources. With virtualization, applications are packaged and run in the form of virtual machines (VM) that share the server infrastructure. Due to the multi-tenant nature of such virtualized data centers, resource management becomes a major challenge for cloud operators to achieve economies of scale. According to a 2010 survey [6], it is the second most concerned problem that CTOs expressed after security. VMs impose extremely diverse resource requirements that need to be accommodated, as they run completely different applications owned by different clients. As such, they are entitled to distinct resource management policies depending on specific needs of their owners.

On the other hand, the infrastructure is managed as a whole by the cloud operator, who relies on a common resource management substrate, and has a wide variety of its own objectives to achieve, such as workload consolidation, cost minimization, and load balancing. Therefore, the resource management substrate must accommodate and orchestrate the needs and interests of both cloud operators and clients. However, current solutions provided by virtualization vendors are far from satisfactory: they are proprietary, hard-coded, and not easily customizable. There exists no interface for cloud clients to express resource management needs in their applications.

In this paper, we present *Anchor*, a new architecture that decouples *policies* from *mechanisms* when it comes to managing resources in the cloud. Stakeholders in the cloud, including both cloud operators and their clients, are able to express and configure their high-level resource management policies, based on performance, cost, and network load, as they deem fit. These policies serve as input to guide mechanisms that manage cloud resources, so that conflicts of interest among stakeholders can be resolved. The output is a mapping between VMs and physical servers: *Anchor* allocates VMs to servers before they are run, and, if necessary, migrates running VMs away from their original hosts using live VM migration. *Anchor* is designed to be scalable to support hundreds of thousands of VMs and servers, to be expressive so that clients and operators can specify a wide variety of their policies and preferences with ease.

To achieve our design objectives with *Anchor*, it may be tempting to formulate the problem as an optimization over certain definitions of utility functions, each reflecting a corresponding policy. However, optimization-based solutions suffer from a number of important deficiencies. *First,* as system-wide performance and costs are optimized, these solutions may not be appealing to the potentially conflicting interests of cloud clients. In this context, the cloud resembles a resource market in which clients and operators are autonomous selfish agents: *individual rationality* needs to be respected for the outcome of the mechanism to be acceptable to all market participants. *Second,* optimization solvers are computationally expensive due to the combinatorial nature of the problem, and do not scale well. The VMware Distributed Resource Scheduler, for example, can only manage up to 32 servers and 1280 VM per cluster [5].

Our design of the *Anchor* architecture is based on a *stable matching* framework from economics theory, which elegantly and efficiently addresses common and conflicting interests of agents in a resource market. In our framework, the concept of *preferences* is used to express various high-level policies that stakeholders specify, and, rather than *optimality*, *stability* is used as the central solution concept of the matching mechanism. Merits of the stable matching framework lie in its competitiveness of outcomes, generality of preferences, efficiency and simplicity of its algorithmic implementations, and most importantly, its overall practicality.

With the design of *Anchor*, this paper has made a number of original contributions. The highlight of our work is a

new multi-stage deferred-acceptance algorithm that matches multiple VMs of heterogeneous resource demands to a single physical server, corresponding to a many-to-one stable matching problem with size heterogeneity. In Sec. II, we present a rigorous treatment of size heterogeneity by clarifying its theoretical ambiguity, proposing a new stability concept, developing algorithms to efficiently find stable matchings with respect to the new stability definition, and prove the convergence and optimality of our algorithms. In the context of stable matching, we then present a simple interface where agents express their policies by expressing how their preferences regarding the opposite side of the market should be constructed. Finally, we present our prototype implementation of *Anchor* on a 20-node server cluster running Oracle VirtualBox [4], as well as a detailed evaluation of its performance with a variety of resource management policies, using both the experimental testbed and large-scale simulations.

II. Stable Matching

Our design objective of the *Anchor* architecture is simple: virtual machines (VMs) with a variety of resource demands need to be matched to a large number of physical servers in modern data centers, and such matching should meet the policies of both clients and operators of cloud services. Resource availability on physical servers, including CPU, memory, storage space, and network bandwidth, is known *a priori*. Each physical server runs a hypervisor to monitor and control all VMs that it hosts. The hypervisor provides a management API for monitoring and controlling VMs.

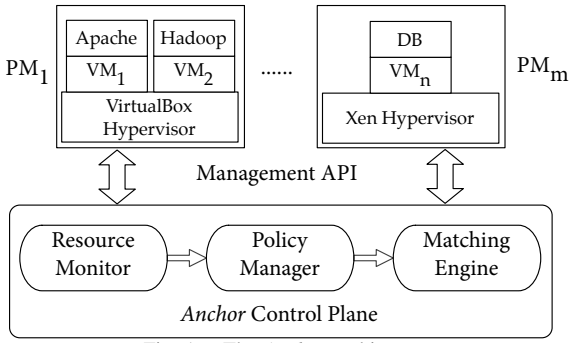

Fig. 1. The *Anchor* architecture.

Each VM is allocated a slice of resource on its hosting server. Throughout the paper, we assume that the size of a slice is a multiple of an *atomic VM*, which represents the smallest amount of resources that a VM can demand. For example, if the atomic VM has a 1 GHz equivalent CPU, 512 MB memory, and 10 GB storage, a VM of size 2 means it has a 2 GHz CPU, 1 GB memory and 20 GB storage. The slicing of resources can be achieved by assigning a weight to the VM. The underlying hypervisor schedules resources proportional to its weight, a feature widely supported by vendors. Note that this may seem an oversimplification of the real-world scenarios, and some may be concerned about the validity of this assumption. We wish to point out that, in practice, such atomic sizing is a common practice among cloud hosting companies such as Amazon and Rackspace to reduce the overhead of managing hundreds of thousands of VMs, and this assumption is thus fairly practical for large-scale public clouds that we focus on. In many related works, similar assumptions in order to reduce the dimensionality of the problem has also been widely adopted [19], [32].

The *Anchor* architecture consists of three main components: a resource monitor, a policy manager, as well as a matching engine, as shown in Fig. 1. The cloud operator configures resource management policies as input to the policy engine. When requests for allocating a new VM or migrating an existing VM arrive, *Anchor* assumes that detailed information about VM configurations and its own policy goals is available at the time. The policy manager then queries information required by both operator and client policies from the resource monitor, which maintains resource usage information through the management API, and translates these high-level objectives into preferences for the VM. It also configures server preferences in a similar manner. These preferences are fed into the matching engine that runs our stable matching mechanism. The result determines the final matching between VMs and servers, and is executed through the management API.

A. The Theory of Stable Matching: A Primer

We start by introducing the basic theory of stable matching in the one-to-one marriage model. In this model, there are two disjoint sets of agents, $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ and $\mathcal{W} = \{w_1, w_2, \ldots, w_p\}$, men and women. Each agent has a transitive preference over individuals on the other side, and the possibility of being unmatched [26]. Preferences can be represented as rank order lists of the form $p(m_1) = w_4, w_2, \ldots, w_i$, meaning that man m_1's first choice of partner is w_4, second choice is w_2 and so on, until at some point he prefers to be unmatched (i.e. matched to the void set). We use $\succ_i$ to denote the ordering relationship of agent i (on either side of the market). If i prefers to remain unmatched than being matched to agent j, i.e. $\emptyset \succ_i j$, then j is said to be *unacceptable* to i, and preferences can be represented just by the list of acceptable partners. Preferences are *strict* if each agent is not indifferent between any two acceptable partners.

Definition 1: An outcome of the market is a *matching* $\mu : \mathcal{M} \times \mathcal{W} \rightarrow \mathcal{M} \times \mathcal{W}$ such that $w = \mu(m)$ if and only if $\mu(w) = m$, and $\mu(m) \in \mathcal{W} \cup \emptyset$, $\mu(w) \in \mathcal{M} \cup \emptyset$, $\forall m, w$.

This implies that the outcome matches agents on one side to those on the other side, or to the empty set. Agents' preferences over outcomes are determined solely by their preferences for their own partners in the matching.

It is clear that we need further criteria to distill a "good" set of matchings from all the possible outcomes. The first obvious criterion is *individual rationality*.

Definition 2: A matching is *individual rational* to all agents of the market, if and only if there does not exist an agent i who prefers being unmatched to being matched with $\mu(i)$, *i.e.,* $\emptyset \succ_i \mu(i)$.

This implies that for an agent that does have a partner in the matching, its assigned partner should rank higher than the

empty set in its preference. Between a pair of matched agents, they are not unacceptable to each other.

The second natural criterion is that a *blocking set* should not occur in a good matching:

Definition 3: A matching μ is *blocked* by a pair of agents (m, w) if they each prefer each other to the partner they receive at μ. That is, $w \succ_m \mu(m)$ and $m \succ_w \mu(w)$. Such a pair is called a *blocking pair* in general.

If there is a blocking pair in the matching, the agents have an incentive to break up and form a new marriage. Therefore such an "unstable" matching is not desirable.

Definition 4: A matching μ is *stable* if and only if it is individual rational, and is not blocked by any pair of agents.

Theorem 1: A stable matching exists for every marriage market.

This can be readily proved by the classic *deferred acceptance algorithm (DA)*, or the *Gale-Shapley algorithm* proposed in [11]. It works by having agents on one side of the market, say men, propose to those on the other side, in order of their preferences. As long as there exists a man, who is free and has not yet proposed to every woman in his preference, he proposes to the most preferred woman who has not yet rejected him, or makes no proposal if no such choice remains. The woman, if free, "holds" the proposal instead of directly accepting it. In case she already has one proposal, she rejects the less preferred. This continues until no proposal can be made, at which point the algorithm stops and matches each woman to the man (if any) whose proposal she is holding. The woman-proposing version works in the same way by swapping the roles of man and woman. It can be readily seen that the order of which men propose is immaterial to the outcome.

The simple marriage model has been extended to more general settings. The *college admissions* problem, where each student seeks to be matched to one college and each college seeks to recruit multiple students, is a well-known many-to-one extension [11]. The *stable roommates* problem is a one-sided variant, where each agent looks for a roommate from a common set [27]. From a practical perspective, due to its efficiency and simplicity, the deferred acceptance algorithm has a profound influence on market design. It has been adopted in a number of practical matching markets, prominent examples of which include the National Resident Matching Program of U.S. for medical school graduates, many medical labor markets in Canada and Britain, and recently school choice systems in Boston and New York City [26]. Due to the richness of the literature on stable matching, it is bold to even attempt a cursory survey of existing results. Instead, the results we have presented are chosen to prepare for a better understanding of our own theoretical development.

B. Models and Assumptions

Resource management in virtualized clouds can be naturally cast as a *stable matching* problem, where the overall pattern of common and conflicting interests between stakeholders can be resolved by confining our attention to outcomes that are stable. Broadly, it can be modelled as a *college admissions* problem [11] where VMs are "students" and servers are "colleges," and they wish to be matched to each other. Preferences can be used as an abstraction of distinct policies, no matter whether they are defined over quantitative measures or qualitative terms.

One may now argue to use optimization that minimizes the total rank sum of the matching as a better alternative. Such an approach is not desirable because preferences embody policies, and the ranking of agents — such as a priority order defined by business contracts — has to be strictly enforced. An optimization solution may match a low priority VM to a machine reserved for high-priority clients, creating a policy violation. Therefore it is not possible to arbitrarily optimize the matching for the sole purpose of minimizing certain metrics.

In the traditional college admissions problem, each college has a quota of the maximum number of students it can take. Unfortunately, this cannot be directly applied to our scenario, as each VM has a different "size," corresponding to its demand for CPU, memory, and storage resources. We cannot simply define the quota of a server as the maximum number of VMs it can take, due to their size heterogeneity.

We formulate VM allocation and migration as a *job-machine stable matching* problem with size heterogeneous *jobs*. Specifically, jobs has different sizes, and machines has different *capacities*. Each machine can accommodate multiple jobs, as long as the total size of jobs does not exceed its capacity. Each job has a transitive preference of all the acceptable machines that have sufficient capacities to hold it. Similarly, each machine has a transitive preference regarding all the acceptable jobs whose size is smaller than its capacity.

This is a more general many-to-one stable matching model, in that the college admissions problem is a special case with jobs of the same size (students). We also note that many other networking problems can be cast into our model. For example, job scheduling in distributed computing platforms such as MapReduce [10] and Hadoop, where jobs submitted from different clients can be of various sizes, and the infrastructure may be privately or publicly shared. Our theoretical results are thus widely applicable to scenarios beyond this paper.

C. Definitions and Challenges

We present theoretical challenges introduced by size heterogeneous jobs in this section. Before doing so, we summarize frequently used notations in our subsequent analysis in Table I.

TABLE I
KEY NOTATIONS IN THE PAPER.

$s(j)$	size of job j
$p(j)$	preference list of job j
$\mu(j)$	job j's assigned machine in the matching μ
$c(m)$	capacity of machine m
$p(m)$	preference list of machine m
$\mu(m)$	machine m's assigned set of jobs in μ
$\mathcal{J}$	set of jobs
$\mathcal{M}$	set of machines

Definitions. Following convention, we can naturally define a *blocking pair* in job-machine stable matching problems based on the following intuition. In a matching μ, whenever a job j prefers a machine m to its assigned machine $\mu(j)$ (can be $\emptyset$ which means it is unassigned), and m has vacant capacity to admit j, or when m does not have enough capacity, but by rejecting some or all of the assigned jobs that rank lower than j, it will be able to admit j, j and m have strong incentive to deviate from μ and form a new matching. Therefore,

Definition 5: A job-machine pair (j, m) is a *blocking pair* if any of the two conditions holds: (1) $c(m) \geq s(j)$, $j \succ_m \emptyset$, and $m \succ_j \mu(j)$. (2) $c(m) < s(j), c(m) + \sum_{j'} s(j') \geq s(j)$, where $j' \prec_m j, j' \in \mu(m)$, and $m \succ_j \mu(j)$.

Depending on whether a blocking pair satisfies condition (1) or (2), we say it is a *type-1* or *type-2* blocking pair.

Definition 6: A job-machine matching is *strongly stable* if it does not contain any blocking pair of job and machine.

For example, in a setting as shown in Figure 2, the matching $A-(a), B-\emptyset$ contains two type-1 blocking pairs (b, B) and (c, B), and one type-2 blocking pair (c, A).

It is clear that both types of blocking pairs are undesirable, and we ought to find a strongly stable matching. However, such a matching may not exist for some problem instances. Figure 2 shows one such example with three jobs and two machines. It can be readily verified that every possible matching contains either type-1 or type-2 blocking pairs.

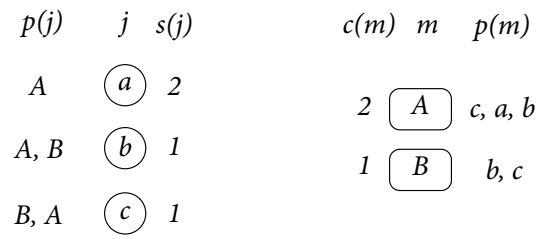

Fig. 2. A simple example where there is no strongly stable matching.

Therefore, we give a weaker definition below.

Definition 7: A matching is *weakly stable* if it does not contain any type-2 blocking pair.

For example in Figure 2, $A-(c), B-(b)$ is a weakly but not strongly stable matching, because it has a type-1 blocking pair (b, A). Thus, weakly stable matchings are a superset that subsumes strongly stable matching. A matching is thus called *unstable* if it is not weakly stable.

Failure of the DA Algorithm. As our first theoretical challenge, how do we find a weakly stable matching, and does it always exist? If we can design an algorithm that produces a weakly stable solution for any given instance, then its existence is clear. It may be tempted to say that the deferred acceptance algorithm (DA) can be readily applied here for this purpose. Jobs propose to machines following the order in their preferences. We randomly pick any free job that has not proposed to every machine on its preference to propose to its current favorite machine that has not yet rejected it. That machine accepts the offer if capacity allows, or capacity is not enough but can be made enough by rejecting several less desirable offers, and rejects it otherwise. Unfortunately, we show that this may fail to be effective.

Figure 3 shows an example similar to Figure 2. Say we first let job a, b, c propose until they cannot. The result is $A-(a), B-(b)$, since b would be rejected by A and c rejected by B. Then we let c propose to A. The result becomes $A-(c), B-(b)$, and a is rejected by A. At this point, only d can propose to A, and A holds the offer. The final result is $A-(c, d), B-(b)$. This is clearly type-2 blocked by (b, A), as A prefers b to d and b prefers A to B.

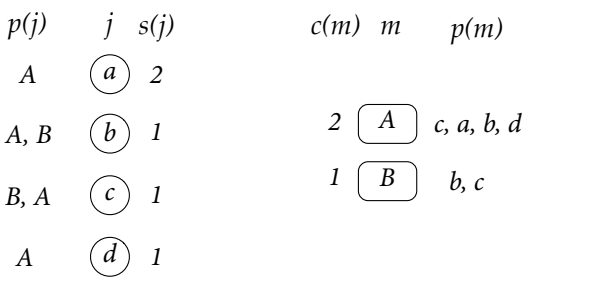

Fig. 3. An example where a possible execution of the DA algorithm produces a type-2 blocking pair (b, A).

On the other hand, if we let d propose to A first before a and b, and keep the rest of execution unchanged, the result becomes $A-(c), B-(b)$, which is weakly stable.

This example demonstrates two problems when applying the DA algorithm to job-machine stable matching. First, the execution sequence is no longer immaterial to the outcome. Second, it may yield an unstable matching. This creates considerable difficulties since we cannot determine which proposing sequence yields the weakly stable matching for an arbitrary problem.

The DA algorithm fails precisely due to the size heterogeneity of jobs. Recall that a machine will reject offers only when its capacity is used up. In the traditional setting with jobs of the same size, this ensures that whenever an offer is rejected, it must be the case that the machine's capacity is used up, and thus any offer made from a less preferred job will never be accepted, *i.e.* the outcome is stable. However, rejection due to capacity is problematic in our case, since a machine's remaining capacity may be increased, and its previously rejected job may be favorable again.

Optimal Weakly Stable Matching. Our second challenge is that, there may be many weakly stable matchings for a given problem. Which one should we choose to operate the system with? Based on the philosophy that cloud operators exist for companies to ease the pain of IT investment and management, rather than the other way around, it is desirable if we can find a *job-optimal* weakly stable matching if it exists, in the sense that every job is assigned its best machine possible in all matchings. For the same reason above, the DA algorithm is again not applicable in this regard, because it may produce type-1 blocking pairs even when the problem admits strongly stable matchings. Thus, our main objective in the following is to design an algorithm that yields the job-optimal weakly

stable matching to serve as the underlying mechanism for the *Anchor* architecture.

D. Algorithms

We first propose a revised DA algorithm, shown in Algorithm 1, that is guaranteed to find a weakly stable matching. The key idea is to ensure that, whenever a job is rejected, any job less preferred than it will not be accepted by the machine, even though the machine has capacity to take it.

The algorithm starts with the set of jobs denoted as $\mathcal{J}$ and the set of machines $\mathcal{M}$. Every machine's offer queue is initialized to be empty, and every job to be free. Then it enters a propose-reject procedure. Whenever there is a free job that has not proposed to every machine, we pick it to propose to the machine at the top of $p(j)$, which contains all the machines that has not yet rejected it. If the machine has sufficient capacity, it holds the offer. If not, it sequentially rejects offers that are less preferable than this one from its queue, in order of its preference, until it can take the offer. If it still cannot take the offer even after rejecting all the less preferable offers, the offer is rejected. Whenever a machine rejects a job, it removes the job, and *all* the jobs ranked lower than this one if any, from its preference. The machine also removes itself from the preferences of its rejected jobs, as it will never accept their offers.

Data: $c(m), p(m), \mu(m), \forall m \in \mathcal{M}$, $s(j), p(j), \mu(j), \forall j \in \mathcal{J}$.
begin
  stop = $False$, $\mu' = \mu$, free = $\mathcal{J}$
  while free.isEmpty() ! = $True$ **do**
    j = free.pop()
    $m = p(j)$.pop()
    if $c(m) \geq s(j)$ **then**
      $\mu(m)$.add$(j), \mu(j) = m, c(m) -= s(j)$
    else
      $j' = \mu(m)$.*last*
      if $j' \succ_m j$ **then**
        free.add$(j), \mu(j) = \emptyset$
      else
        while $j' \neq \emptyset, j' \prec_m j, c(m) < s(j)$ **do**
          $\mu(m)$.remove$(j'), c(m) += s(j')$
          $\mu(j') = \emptyset$, free.add(j')
          best_rejected = j'
          $j' = \mu(m)$.last()
        if $c(m) \geq s(j)$ **then**
          $\mu(m)$.add$(j), \mu(j) = m, c(m) -= s(j)$
        else
          free.add(j), best_rejected = j'
        for $j \in p(m), j \prec_m$ best_rejected **do**
          $p(j)$.remove$(m), p(m)$.remove(j)
return $\mu, c(m) \forall m \in \mathcal{M}$

Algorithm 1: *The revised deferred acceptance algorithm.*

A pseudo-code implementation is shown in Algorithm 1. We can readily see that the order in which free jobs propose in Algorithm 1 is immaterial to the outcome, similar to the original DA algorithm. Moreover, we can prove that the revised DA algorithm guarantees that type-2 blocking pairs will never occur in the result.

Lemma 1: Algorithm 1, in any execution order, produces a unique weakly stable matching among the set of jobs $\mathcal{J}$ and the set of machines $\mathcal{M}$.

Proof: The proof of uniqueness is essentially the same as that for the DA algorithm in the seminal paper [11]. We prove the weak stability of the outcome by contradiction. Suppose that Algorithm 1 produces a matching μ with a type-2 blocking pair (j, m), i.e. there is at least one job j' worse than j to m in $\mu(m)$. Since $m \succ_j \mu(j)$, j must have proposed to m and been rejected. When j was rejected, all the jobs with ranking lower than j is ensured to be removed from m's preference $p(m)$. Thus m will not accept any job worse than j. Thus, $j' = \emptyset$. This contradicts with the assumption. ■

This also proves the existence of weakly stable matchings, as the revised DA algorithm clearly terminates within $O(|\mathcal{J}|^2)$.

Theorem 2: Weakly stable matchings always exist for job-machine matching problems.

Naturally, the revised DA algorithm will still produce type-1 blocking pairs, and the result may not be the job-optimal weakly stable matching as we have defined in Sec. II-C. In order to find the job-optimal matching, an intuitives idea is to run the revised DA algorithm multiple times, each time with those type-1 blocking jobs proposing to their desirable machines that form blocking pairs with them. The intuition is that, type-1 blocking jobs can be possibly improved at no expense of other jobs. However, simply doing so may make the matching unstable, because when a machine has both type-1 blocking jobs leaving from and proposing to it, it may have more capacity available to take jobs better than those it accepts according to its capacity before the jobs leaving.

To give an example, let us take a look at the problem in Figure 4. We now run the revised DA algorithm over this example. The result will then be $A-(d), B-(e), C-(a)$. Clearly there are two type-1 blocking pairs, (a, A) and (b, C). Say we fix this by letting a propose to A and b propose to C. Then we have a new type-2 blocking pair (c, C) due to the removal of job a from C, where c prefers C to being unassigned, and C prefers c to b and by rejecting b it has enough capacity to admit c. This is a result of C wrongly accepting b when it actually has more capacity after a leaves.

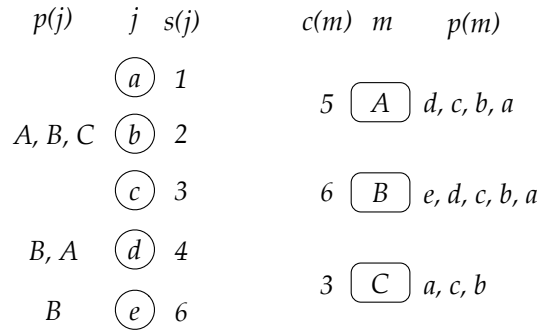

Fig. 4. An example where simply running the revised DA algorithm multiple times will produce a new type-2 blocking pair (c, C).

We now design a *multi-stage deferred acceptance* algorithm to iteratively find a better weakly stable matching with respect to jobs. The algorithm proceeds in stages. Whenever there is a type-1 blocking pair (j, m) in the result of previous stage μ_{t-1}, the algorithm enters the next stage where the blocking

machine m will accept new offers. The blocking job j is removed from its previous machine, so that it can make new offers to machines that have rejected it in previous stages. The machine $\mu_{t-1}(j)$'s capacity is also updated accordingly. Moreover, to account for the effect of job removal, all jobs that can potentially form type-1 blocking pairs with j's previous machine $\mu_{t-1}(j)$ if j leaves (there may be other machines that j form type-1 blocking pairs with) are also removed from their machines and allowed to propose in the next stage. This ensures that the algorithm does not produce new type-2 blocking pairs during the course, as we shall prove soon. In each stage, we run the revised DA algorithm with the selected set of proposing jobs $\mathcal{J}'$ possible to improve their matching, and the entire set of machines with updated capacity $c_t^{pre}(m)$. The entire procedure is shown in Algorithm 2.

Data: $c(m), p(m), \forall m \in \mathcal{M}$, $s(j), p(j), \forall j \in \mathcal{J}$.
begin
  $\mu_0 = \emptyset, t = 0, \text{stop} = False, \mathcal{J}' = \emptyset$.
  while stop $== False$ **do**
    $t = t + 1, \mu' = \mu_{t-1}$
    for $m \in \mathcal{M}$ **do** $c_t^{pre}(m) = c_{t-1}(m)$
    while $\Omega =$ find_type-1(μ', c_t^{pre}, $\mathcal{J}'$) **do**
      for $j \in \Omega$ **do**
        if $\mu'(j)! = \emptyset$ **then**
          $c_t^{pre}(\mu'(j)) += s(j)$
          $\mu'(s)$.remove(j), $\mu'(j) = \emptyset$
          $\mathcal{J}'$.add(j)
        else
          $\mathcal{J}'$.add(j)
    if $\mathcal{J}' == \emptyset$ **then**
      break
    $(\mu_t, c_t(m)) =$ RevisedDA($c_t^{pre}(m), p(m), s(j), p(j), \mu', \mathcal{J}'$)
    if $\mu_t == \mu_{t-1}$ **then**
      stop $= True$
return μ_t

Algorithm 2: *The multi-stage deferred acceptance algorithm.*

E. Analysis

We now rigorously prove key properties of Algorithm 2: its correctness, convergence, and job-optimality.

Correctness. First we establish the correctness of our algorithm.

Theorem 3: There is no type-2 blocking pair in the matchings produced at any stage in Algorithm 2. For a given problem, Algorithm 2 produces a unique weakly stable matching in each stage, no matter what the execution order is.

Proof: This can be proved by induction. As the base case, we already proved that there is no type-2 blocking pair after the first stage of Algorithm 2 in Lemma 1.

Given there is no type-2 blocking pair after stage t, we need to show that after stage $t+1$, there is still no type-2 blocking pair. Suppose after $t+1$, there is a type-2 blocking pair (j, m), *i.e.,* $c_{t+1}(m) < s(j)$, $c_{t+1}(m) + \sum_{j'} s(j') \geq s(j)$, where $j' \prec_m j, j' \in \mu_t(m), m \succ_j \mu_{t+1}(j)$. If $c_{t+1}^{pre}(m) \geq s(j)$, then by Algorithm 2 j must have proposed to m and been rejected. Thus it is impossible for m to accept any job j' less preferable than j in $t+1$.

If $c_{t+1}^{pre}(m) < s(j)$, then j did not propose to m in $t+1$. Since there is no type-2 blocking pairs after t, j' must be accepted in $t+1$. Now since $c_{t+1}^{pre}(m) < s(j)$, the sum of the remaining capacity and total size of newly accepted jobs after $t+1$ must be less than $c_{t+1}^{pre}(m)$, *i.e.* $c_{t+1}(m) + \sum_{j''} s(j'') \leq c_{t+1}^{pre}(m) < s(j)$, where j'' denotes the newly accepted jobs in $t+1$. This contradicts with the assumption that $c_{t+1}(m) + \sum_{j'} s(j') \geq s(j)$ since $\{j'\} \subseteq \{j''\}$.

If $c_{t+1}^{pre}(m) = 0$, then m only has jobs leaving from it. Since there is no type-2 blocking pair after t, clearly there cannot be any type-2 blocking pair in $t+1$.

Therefore, type-2 blocking pairs do not exist in any stage of Algorithm 2. The uniqueness of the matching result at each stage is readily implied from Lemma 1. The lemma holds. ■

Convergence. Next we prove the convergence of Algorithm 2. The key observation is that our multi-stage algorithm produces a weakly stable matching at least as well as that at the previous stage from the job perspective.

Lemma 2: At any consecutive stages t and $t+1$ of Algorithm 2, $\mu_{t+1}(j) \succeq_j \mu_t(j), \forall j \in \mathcal{J}$.

Proof: This is a direct result of the algorithm design, since in $t+1$ every proposing job proposes to machines that have previously rejected it. If any of these machines accepted it, $\mu_{t+1}(j) \succ_j \mu_t(j)$. If none of these machines accepted it, it will for sure be able to propose to its previous machine $\mu_t(j)$ since $c_{t+1}^{pre}(\mu_t(j))$ must be no smaller than $s(j)$. $\mu_t(j)$ will for sure accept j at $t+1$, because it will only receive offers from jobs that it previously rejected, possibly also from other jobs that it previously accepted if they propose to other machines and are rejected in $t+1$. j remains favorable to m, even when all of m's accepted jobs in t proposed to m in $t+1$ again. ■

Therefore, Algorithm 2 always tries to improve the weakly stable matching it found in the previous stage, whenever there is such a theoretical possibility suggested by the existence of type-1 blocking pairs. However, from Lemma 2 it is possible that a job's machine at $t+1$ remains the same as in the previous stage. In fact, it is possible that the entire matching is the same as the one in previous stage, i.e. $\mu_{t+1} = \mu_t$. This can be easily verified by using the example of Figure 2. After the first stage, the weakly stable matching is $A-(c), B-(b)$. First b wishes to propose to A in the second stage. Then we assign b to $\emptyset$ and B has capacity of 1 again. c then wishes to propose to B too. After we remove c from A and update A's capacity, a now wishes to propose to A. Thus at the next stage, the same set of jobs a, b, c will propose to the same set of machines with same set of capacity, and the result will be the same matching as in the first stage. In this case, Algorithm 2 will terminate with the final matching that it cannot improve upon as its output. We thus have:

Theorem 4: Algorithm 2 always terminates in finite time.

Job-Optimality. We can rigorously prove the job-optimality of the weakly stable matching that Algorithm 2 produces. That is, no matter how the algorithm terminates, it always produces the job-optimal weakly stable matching as its output.

Theorem 5: Algorithm 2 always produces the job-optimal weakly stable matching when it terminates, in the sense that every job is at least as well off in the weakly stable matching produced when the algorithm terminates as it would be in any

TABLE II
Anchor'S POLICY INTERFACE.

Functionality	*Anchor* API Call
create a operator policy group	g_o = create('operator')
create a client policy group	g_c = create('client')
add/delete server to/from a group	add/delete(g_o, s)
add/delete VM to/from a group	add/delete(g_c, v)
set ranking factors	set_factors(g, factor1, ...)
set placement constraints	limit(g_c, server_list)

other weakly stable matching.

Proof: Algorithm 2 terminates at stage t when either there is no type-1 blocking pair, or there is at least one type-1 blocking pair but $\mu_t = \mu_{t-1}$. For the former case, we show that our algorithm only permanently rejects jobs from machines that are impossible to accept them in all weakly stable matchings, when the jobs cannot participate any further. The resulting assignment is therefore optimal. For the latter case, we can also show that it is impossible for jobs that participated in t to obtain a better machine. A detailed proof is postponed to the Appendix of this paper. ■

Finally, we present another fact regarding the outcome of our algorithm.

Theorem 6: For a given problem instance, Algorithm 2 produces a unique job-optimal strongly stable matching when it terminates with no job proposing.

Proof: This can be easily proved by contradiction. Assume that the matching produced when the algorithm terminates is not strongly stable. Thus, we must be able to find a type-1 blocking pair, say (j, m), as implied by Lemma 3. m will participate in the next stage, and j will be willing to propose to m, and our algorithm will continue to run, rather than terminating. This contradicts with our assumption. ■

We conjecture that when the algorithm terminates with type-1 blocking pairs, the problem does not admit a strongly stable solution. The proof is, however, not immediate and is left for future work.

III. RESOURCE MANAGEMENT POLICIES

As the underlying mechanism of *Anchor*, we have just presented two new algorithms to produce a stable matching between multiple VMs of various sizes, as *jobs*, and their physical servers, as *machines*, given their preferences. We now introduce the policy engine in the *Anchor* architecture, which constructs preference lists according to various resource management policies. The cloud operator and clients interact with the policy engine through its API, as shown in Table II.

In order to reduce the overhead of resource management, we use *policy groups* in the design of the policy engine. They can be created with the create() API call for both the operator and the clients. Each policy group contains a set of servers or VMs that are entitled to one specific policy. The policy is configured by the set_factors() call that informs the policy engine what the factors to be considered are, in descending order of importance. *Anchor* then performs a multi-pass sorting procedure, first based on the least important factor, then the second least, and so on, to produce preferences that adhere to the policy. With policy groups, only a common preference list is needed for all members in the group whenever possible. Membership of policy groups is maintained by add() and delete(). The limit() call is used to set placement constraints as we will discuss in Sec. III-C.

In practice, it is possible for the cloud operator to configure policies on behalf of its clients, in case they do not explicitly specify any. This is indicated by enrolling clients in the *default* policy group.

A. VM Allocation

We begin our examination from the perspective of cloud operators, who rely on *Anchor* to optimally assign VMs to servers before any cloud service can run. What they consider "optimal" can differ substantially: some may seek to consolidate the workload, some may wish to balance the load across servers to handle time-varying dynamics, and some may even employ a more sophisticated cost model that incorporates energy costs. We show that these preferences can all be expressed and realized using the policy API of *Anchor*.

Server consolidation/packing. Cloud operators usually wish to consolidate the workload by packing VMs into a small number of highly utilized servers, so that idle servers can be powered down to conserve operational costs. To realize this policy through *Anchor*, servers should be configured to prefer a VM with a larger size, since the operator wishes to maximize utilization. This can be done with a call to set_factors (g_o, -vm_size) or set_factors(g_o, 1/vm_size). For VMs that belong to the default policy group, their preferences are ranked in the descending order of server load. One may use the total size of active VMs as the metric of load (set_factors (g_c, 1/server_utilization)). Alternatively, the total number of active VMs can also serve as a heuristic metric (set_factors (g_c, 1/num_of_vm)).

Notice that consolidation is closely related to packing, and the above configuration also resembles the *first fit decreasing* heuristic widely used to solve packing problems by iteratively assigning the largest item to the first bin that fits.

Memory consolidation. Other techniques to consolidate load can also be incorporated into *Anchor*. In-memory page sharing [22], [30], [33], for example, is a recently proposed methodology that consolidates the memory footprint of VMs. When a set of similar VMs (*e.g.*, with identical guest operating systems) are running on the same server, a large number of memory pages can be identical. Page sharing searches for identical pages and maintains only a shared copy among a set of VMs.

Due to privacy or confidentiality concerns, cloud operators are only able to identify the type of guest operating systems that client VMs run, and may not have access to more detailed information about applications running inside VMs. Based on observations made in previous work [22], [30], we assume that a VM's page sharing potential on a server can be qualitatively ranked, and such ranking is affected first by the sharing potential of its guest operating system,

represented by vm_os_sharing (higher values imply that more aggressive sharing is feasible), and then by the size of the VM. In this case, an operator can use set_factors(g_o, 1/vm_os_sharing, 1/vm_size) to incorporate in-memory page sharing and to consolidate memory utilization.

Load balancing. Another popular resource management policy is load balancing, which distributes VMs across all the servers to mitigate performance degradation due to application dynamics over time. This can be seen as the opposite of consolidation. In this case, we can configure the server preference with a call to set_factors(g_o, vm_size), implying that servers prefer VMs of smaller sizes. VMs in the default policy group are configured by a call to set_factors(g_c, server_utilization), such that they prefer less utilized servers. This can be seen as a *worst fit increasing* heuristic. Another possible load balancing metric is server temperature. A lower temperature in a server typically implies that the server is lightly loaded. If this is the case, set_factors(g_c, server_temperature) can be used to configure VM preferences.

From the perspective of cloud clients, other than choosing to join the default policy group and follow the operator's configuration, cloud clients can also express their unique policies.

Resource hunting. Depending on the resource demand of applications, VMs can be CPU, memory, or bandwidth-bound, or even resource-intensive along a combination of multiple resource dimensions. Though resources are sliced into fixed slivers, most modern hypervisors support the dynamic resizing of VMs, as long as there are resources available at the server. For example, if a VM is allocated 50% CPU on a server when its workload increases, the hypervisor may allow a temporarily burst to beyond 50%, provided that such a burst does not affect other co-existing VMs. With respect to memory, with a technique known as *memory ballooning*, the hypervisor is able to dynamically reduce the memory footprints of idle VMs, so that memory allocated to more heavily loaded VMs can be increased.

With servers supporting these features, clients may wish to configure their policies according to the resource usage pattern of their VMs, which is unknown to the operator. CPU-bound VMs can be added to a *CPU-bound* policy group, which is configured with a call to set_factors(g_c, 1/server_freecpu). Their preferences are then ranked in the descending order of the server's available CPU cycles. Similarly, memory-bound and bandwidth-bound policy groups may be configured with the call set_factors(g_c, 1/server_freemem) and set_factors(g_c, 1/server_freebw), respectively. If a client wishes to jointly consider resources along multiple dimensions, say, both CPU and memory, a call to set_factors(g_c, 1/server_freemem, 1/server_freecpu) implies that preferences of VMs in this policy group are ranked first by server's available memory, and then by available CPU cycles.

B. VM Migration

With support for live migration of virtual machines across the boundary of servers in existing products such as XenMotion [9], the mapping between VMs and servers can be updated to better balance the workload, or to perform server maintenance. The VM migration policies for both the cloud operator and its clients can be configured so that they are similar to their VM allocation policies, with a focus on performance.

However, rather than solely considering the performance after migration completes, the performance *during* the migration process may also be an important concern to both the operator and its clients, since live VM migration involves transferring memory states of running VMs across a network.

Network-aware migration. Though there are a number of new data center network designs proposed in the literature [13], [15], [23], to a large extent, production data center networks still follow a three-tier tree architecture as shown in Figure 5. Packets transferred within the same subnet traverse only the router in the access tier, with no load to the upper tier routers. Traffic across the boundary of subnets, in contrast, has to traverse more hierarchies and is more costly. Live migration of VMs requires bulk transfers of disk images that are bandwidth-intensive [32], yet we wish to minimize the imposed traffic footprint as data center networks are already fraught with scalability and efficiency issues [13], [15]. Thus it is of the cloud operator's interest to be *network-aware*, and migrate VMs to servers in the topological vicinity, which mitigates the traffic load on the network. The cloud operator also needs to be concerned with the sizes of VM memory images, as they determine the migration overhead on the network.

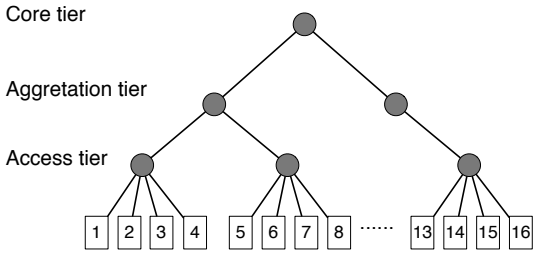

Fig. 5. A three-tier data center network topology.

On the other hand, cloud clients have their own considerations. They are more concerned with available bandwidth on servers, since they wish to minimize the time required for the migration process [9]. This reflects a conflict of interest between the operator, who prefers to minimize load on its network, and its clients, who prefers to minimize migration time.

Anchor relies on the concept of *stability* to resolve such a conflict. The operator can set its policy by set_factors(g_o, num_of_hops, vm_memsize), implying that servers prefer to minimize the migration overhead by migrating VMs over a smaller number of hops in the network, and by preferring those with a smaller memory footprint. Clients may configure their

migration policies with set_factors (g_c, 1/server_freebw), implying that they prefer to migrate their VMs to servers with more available bandwidth.

Energy minimization. Energy constitutes a major part of the operator's costs [12]. To minimize energy consumption, a recent study proposed new methodologies based on a workload analysis of a production data center [29]. The idea is to measure the workload correlation of VMs, and identify those whose workloads do not peak at the same time. By dynamically migrating and consolidating VMs with uncorrelated workloads, servers can be put into low-energy state more often without violating individual client's SLA. Such a strategy can be readily implemented by having the resource monitor collect and compute workload correlation statistics, and setting the operator's policy group by set_factors(g_o, vm_correlation). Each server's preference is then ranked in an ascending order of the workload correlation between its existing VMs and the target VM to be migrated.

C. Additional Examples

With examples we have illustrated so far, it is evident that the policy engine in *Anchor* is very flexible when it comes to expressing the needs and preferences of both the operator and its clients. Such versatility can be further shown in the following two examples, which are applicable in both scenarios of VM allocation and migration.

Tiered service. It is common practice to implement tiered services in an operational cloud, by associating each VM with its priority class. For example, Amazon EC2 offers three tiers of VMs to use: *reserved*, *on-demand*, and *spot* instances, with a descending order of priority and price. It is straightforward to incorporate tiered services when configuring operator policies in *Anchor:* one just needs to configure servers so that they prefer VMs with higher priority classes, with a call to set_factors(g_o, priority).

Incomplete preferences. Due to upgrades and cost concerns, a data center typically contains several generations of servers in operation at the same time. It is therefore possible that some VMs can only be placed on a subset of servers, due to hardware constraints. Such placement limitations can be accommodated in *Anchor*, as the list of preferences does not need to contain a complete set of servers. The policy engine in *Anchor* supports an additional API call limit(g_c, server_list) so that VM preferences in a client policy group g_c explicitly exclude any server outside of server_list.

Dynamically changing preferences and VM sizes. It is common that the servers' and VM's preferences may change over time. *Anchor* also supports dynamically changing preferences and VM sizes. Whenever such a change occurs, the corresponding party automatically leaves its current partner and becomes unassigned, which yields a type-1 blocking pair. The matching then becomes unstable, and the multi-stage DA algorithm as shown in Algorithm 2 can readily run to produce a new stable matching with respect to the updated preferences and VM sizes.

IV. Implementation and Evaluation

We are now ready to investigate the effectiveness and performance of the *Anchor* architecture. We begin with a prototype implementation of *Anchor* with about 1500 LOC in the Python language. Our implementation is based on Oracle VirtualBox 3.2.10 [4].

The resource monitor is implemented as a Python application that maintains resource statistics of servers and VMs using SQLite, a lightweight database engine. The sqlite3 Python bundle is utilized to update records in the database. The resource monitor periodically listens for usage reports — once per second — from a daemon in each physical server, which we have implemented for the sole purpose of collecting measurements of CPU and memory usage, via the VirtualBox management API (VBoxManage metrics). Our daemon utilizes the iptraf tool to detect the available bandwidth of each server, since VirtualBox does not provide suitable APIs for this purpose.

The policy manager also utilizes SQLite to manage policy groups for both the operator and cloud clients, and updates its databases upon receiving an *Anchor* API call. For efficiency, it maintains all the preferences in memory. When an allocation or migration request arrives, it obtains necessary information from the resources monitor's database, and sends sorted preferences to the matching engine.

The matching engine implements both the revised and the multi-stage deferred acceptance algorithms in Python. We preprocess server preferences (with a complexity of $O(n)$) to obtain an inverse of the list indexed by rank. Each subsequent rank comparison can thus be performed in constant time. For maximum efficiency, we use numpy in Python to implement the data structure of preferences. The matching engine utilizes the VirtualBox management API to start or migrate VMs according to matching results.

Our evaluation of Anchor is based on a prototype data center consisting of 20 Dual Dual-Core Intel Xeon 3.0 GHz machines connected over gigabit ethernet. Each machine has 2 GB memory. Thus, we define the atomic VM of size 1 to have 1.5 GHz CPU and 256 MB memory. Each server has maximum capacity of 7 in terms of atomic VM (since the hypervisor also consumes server resources). All machines run Ubuntu 8.04.4 LTS with Linux 2.6.24-28 server. A cluster of Dual Intel Xeon 2.4 Ghz servers are used to generate workload for some of the experiments. One node in the cluster is designated to run the *Anchor* control plane, while others host VMs. Our VMs, if not otherwise noted, run Ubuntu 8.10 server with Apache 2.2.9, PHP 5.2.6, and MySQL 5.0.67.

Results from our experimental evaluation are presented in four categories. Sec. IV-A investigates the effectiveness of *Anchor* on enforcing policies. Sec. IV-B uses the resource hunting policy for clients to evaluate the efficiency of resource allocation in *Anchor*. Sec. IV-C evaluates *Anchor*'s ability to resolve conflicts of interest between cloud operators and clients, and Sec. IV-D shows the scalability and practicality of our algorithms using simulations at larger scales.

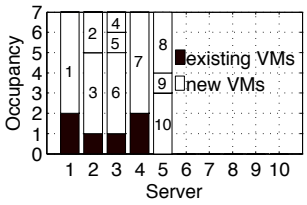

Fig. 6. Consolidation.

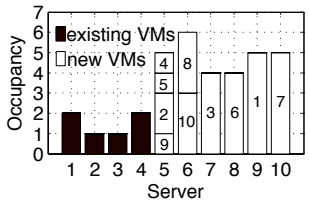

Fig. 7. Load Balancing.

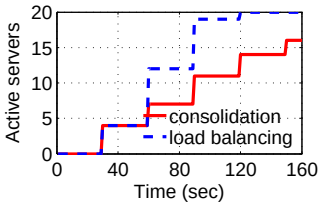

Fig. 8. Time series of the number of active servers.

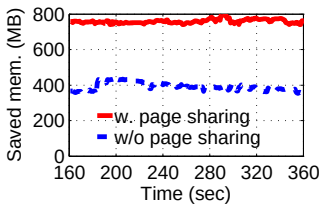

Fig. 9. The effect of page sharing.

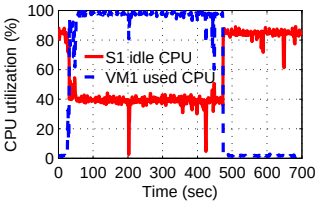

Fig. 10. VM1 CPU usage on S1.

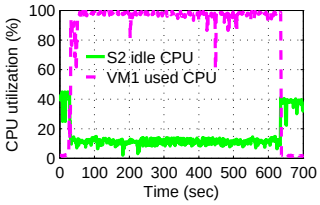

Fig. 11. VM1 CPU usage on S2.

A. *Effectiveness of Realizing Policies*

Our first set of experiments uses both consolidation and load balancing policies for VM allocation to demonstrate the effectiveness of *Anchor* in realizing resource management policies specified by its clients. We assume that clients follow the operator's default policy in all our experiments in this category, so there is no conflicting interest involved.

The first experiment is to use *Anchor* to allocate 10 VM to 10 servers, the first four of which are configured with an utilization of 2, 1, 1, 2, respectively. Figure 6 shows the result of using the consolidation policy, where VM preference is ranked in descending order of server utilization, and server preference is ranked in descending order of VM size. We observe that all the VMs are packed into the first five servers, whose utilization are thus maximized. On the other hand, the load balancing policy strips the VMs across the idle servers, resulting in a more balanced server load as shown in Figure 7. Algorithm 2 takes 3 iterations to converge to the strongly stable matching of Figure 6 for the former case, and 2 iterations for the latter.

We then experiment a more complex scenario where 5 allocations are performed at 30, 60, 90, 120, and 150 seconds, respectively, each time with 10 VMs. Each server is configured to be empty before the experiment. Figure 8 shows a time series comparison of the number of active servers resulted from using the two policies in this scenario. Clearly, the consolidation policy consistently requires less number of servers except for the first allocation, the result of which is identical since each server is initially empty in both cases.

Finally, we incorporate page sharing potential into servers' preferences, *i.e.,* we configure them with set_factors(g_o, 1/vm_os_sharing, 1/vm_os) as in Sec. III-A. The same set of 5 allocations are performed as in the last experiment. Since VirtualBox supports page sharing only for Windows guests, we configure the second and third batches of VMs to run Windows XP 64-bit instead of Ubuntu. Figure 9 shows the memory consolidated by page sharing averaged over servers, which is calculated by multiplying the number of Windows VMs and total memory shared between them. We observe that when page sharing potential is considered, on average around 400 MB more memory is saved on each server, translating to a 100% improvement over a simple consolidation policy.

Result: Anchor *effectively realizes the desired resource management policies as specified by the cloud operator.*

B. *Efficiency of Resource Allocation*

We then evaluate the efficiency of *Anchor* resource allocation, by allowing clients to use resources hunting policies in Sec. III-A. We enable memory ballooning in VirtualBox to allow the temporary burst of memory use. CPU-bound VMs are configured to run a 20 newsgroups Bayesian classification job with 20,000 newsgroups documents, based on the *Apache Mahout* machine learning library [1]. Memory-bound VMs run a facebook-like Web 2.0 application called *Olio* that allows users to add and edit social events and share with others [3]. Its MySQL database is loaded with a large amount of data so that performance is memory critical. We use *Faban*, a benchmarking tool for tiered web applications, to inject workload and measure Olio's performance [2].

Our first experiment comprises of 2 servers (S1, S2) and 2 VM (VM1 and VM2). S1 runs a memory-bound VM of size 5, and S2 runs a CPU-bound VM of the same size before allocation. VM1 is CPU-bound with size 1 while VM2 is memory-bound with size 2. Assuming servers adopt a consolidation policy, we run *Anchor* first with the resources hunting policy, followed by another run with the default consolidation policy for the two VMs. In the first run, *Anchor* matches VM1 to S1 and VM2 to S2, since VM1 prefers S1 with more available CPU and VM2 prefers S2 with more memory. In the second run, *Anchor* matches VM2 to S1 and VM1 to S2 by virtue of consolidation.

We now compare CPU utilization of VM1 in these two matchings as shown in Figure 10 and 11, respectively. From Figure 10, we can see that as VM1 starts the learning task at around 20 sec, it quickly hogs its allocated CPU share of 12.5%, and bursts to approximately 40% on S1 (80%-40%). Some may wonder why it does not saturate S1's CPU. We conjecture that the reason may be related to VirtualBox's implementation that limits CPU resources allocated to a single VM. In the case it is matched to S2, it can only consume up to about 30% CPU, while the rest is taken by S2's existing VM as seen in Figure 11. We also observe that the learning task takes about 600 sec to complete on S2, compared to 460 sec on S1, a performance penalty of 30%.

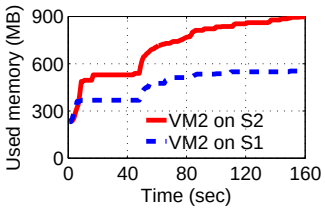

Fig. 12. VM2 memory usage on S1 and S2.

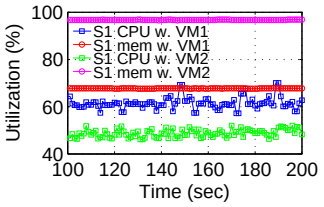

Fig. 13. S1 CPU and memory usage.

We then look at the memory-bound VM2. Figure 12 shows a time series of memory allocation comparison between the two matchings. Recall that VM2 has size 2, and should be allocated 512 MB memory. By the resources hunting policy, it is matched to S2, and obtains its fair share as soon as it is started at around 10 sec. When we start the Faban workload generator at 50 sec, its memory allocation steadily increases as an effect of memory ballooning to cope with the increasing workload. At steady state it utilizes about 900 MB memory. On the other hand, when it is matched to S1 by the consolidation policy, it only has 400 MB memory after startup. The deficit of 112 MB is allocated to the other memory hungry VM that S1 is running before the allocation. VM2 gradually reclaims its fair share as the workload of Olio database rises, but cannot get any extra resource beyond that point.

We also compare Faban benchmark results for Olio in the two matchings. Table III shows Olio's 90-th percentile response time of various operations measured by Faban. A significant performance improvement is observed when VM2 is matched to S2 in all categories. Specifically, for the memory-intensive TagSearch and EventDetail operations, the response times are two orders of magnitude shorter. This clearly demonstrates the advantage of allowing clients to express policies specific to their resource usage pattern compared to not doing so.

Client resource hunting policy serves also to the benefit of the operator and its servers. Figure 13 shows S1's resource utilization during the period [100, 200] sec. When resource hunting policy is used, *i.e.* when S1 is assigned VM1, its total CPU and memory utilization are aligned at around 60%, because VM1's CPU-bound nature is complementary to the memory-bound nature of S1's existing VM. However, when S1 is assigned the memory-bound VM2 by the consolidation policy, its memory utilization surges to nearly 100% while CPU utilization lags at only 50%. A similar observation can be made for S2.

Result: Anchor *enables efficient resource utilization of the infrastructure and improves performance of its VMs, by allowing individual client to express policies specific to its resource needs through its flexible API.*

C. Resolving Conflicting Interests

We now conduct experiments to evaluate *Anchor* in addressing the conflicts of interest between operator and clients. We consider a migration scenarios where servers and VMs have distinct and possibly conflicting goals.

TABLE III
OLIO'S 90TH% RESPONSE TIME COMPARISON (IN SEC).

Type	On S2	On S1	Required
HomePage	0.2	9.2	1.0
Login	0.12	4.7	1.0
TagSearch	0.42	>19.8	2.0
EventDetail	0.28	>19.8	2.0

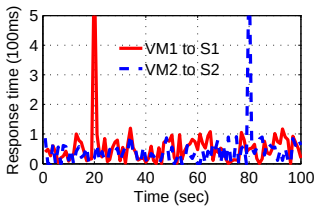

Fig. 14. *Anchor* based on stable matching migrates VM1 to S1 and VM2 to S2.

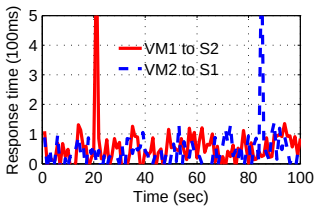

Fig. 15. Optimization migrates VM1 to S2 and VM2 to S1.

Our setup involves two VMs to be migrated, VM1 and VM2, and two servers S1 and S2 as the potential destination. Both VMs are initially hosted at the same server, whose hop distance to S1 and S2 is 1 and 2, respectively. VM1 has 512 MB memory, and VM2 768 MB. Both servers are configured to have a free capacity of 3. To emulate server bandwidth discrepancy, we use the default token bucket implementation in Linux (tc) to limit the bandwidth of S1 to 500 Mbps, and that of S2 to 800 Mbps. VMs' preferences are configured with a call to set_factors(g_c, 1/servers_freebw), and servers' by set_factors(g_o, num_of_hops, vm_memsize) as in Sec. III-B.

With the above setup, *Anchor* matches VM1 to S1 and VM2 to S2 by Algorithm 2. We compare this stable outcome with the result of an optimization that minimizes total utility of the matching. Utility here is defined for a pair of VM and server by (vm_size $\times$ num_of_hops) / server_freebw, in order to jointly consider the interest of operator and clients. This can be seen as the processing time needed for the network to migrate the VM to the given server, and the optimization can then be regarded as a minimum weighted bipartite matching that minimizes the total processing time of migration. This leads to the matching of VM1-S2 and VM2-S1, as can be obtained from calculating and comparing the total utility of all possible matchings.

We then conduct a performance comparison from the cloud clients' perspective. We run the Faban workload generator during the course of migration on both VMs, and plot the response time for displaying Olio homepage as the performance indicator. The migration of VM1 starts at 10 sec, and that of VM2 starts at 60 sec. As shown in Figure 14 for *Anchor*, at 20 sec there is a 1-sec burst of response time. This corresponds to the last stage of live migration where the VM is suspended with the delta of its memory pages since the last transfer being sent to the target server. In total it takes 11 sec to migrate VM1 to S1. A 1-sec down time is also observed for VM2, and it takes 21 sec to migrate VM2 to S2. For the matching produced

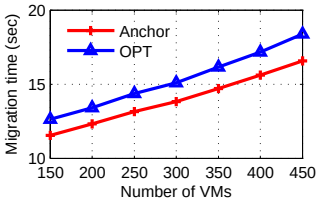

Fig. 16. *Anchor* outperforms OPT in migration time.

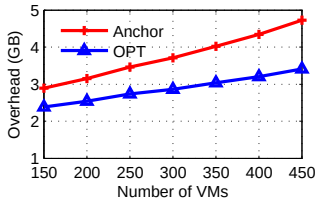

Fig. 17. OPT outperforms *Anchor* in overhead.

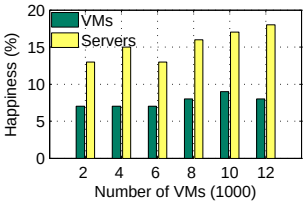

Fig. 18. Happiness.

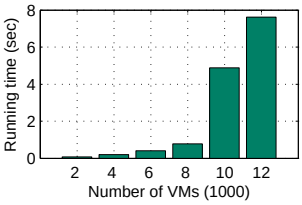

Fig. 19. Running time.

by the optimization, it takes 26 sec to migrate VM2 to S1 as shown in Figure 15, which is about 25% worse. Thus from VM perspective, *Anchor* based on stable matching is favorable to optimization.

We also compare migration performance from the operator's perspective. We measure the total amount of traffic during migration by using the /proc/net/dev interface on the source server, and multiply the total bytes sent by the hop distance (recall both VMs are hosted in the same source server). Though this includes some background traffic, its effect is negligible compared to the bulky migration traffic. We observe that optimization results in a total of 2342 MB overhead, about 12% less than the stable matching approach of *Anchor* which is 2629MB.

The reason for the observed performance discrepancy is that, optimization tries to "arbitrate" the conflicting interest of the operator and its clients by considering all the factors in its utility metric. *Anchor*, on the other hand, uses the stability concept to resolve this issue. The VM-proposing Algorithm 2 yields the VM-optimal stable outcomes as proved in Sec. II-E. Therefore the overall performance is favorable to VMs compared to that of optimization.

To further validate this intuition, we conduct simulations based on the three-tier data center network topology of Figure 5. The quota of each server is uniformly distributed in $[1, 7]$. The hop distance between servers is generated by assuming the fan-out of access routers to be 4, and the fan-out of aggregation routers to be 8. The initial placement of VMs on servers is random. The size of VM's memory image is uniformly distributed in $[256, 1024]$ MB, and the available server bandwidth is uniformly distributed in $[0.5, 1.5]$ Gbps. The results, as shown in Figure 16 and 17, are consistent with our experiments: *Anchor* offers 15% shorter migration time on average over 100 runs, while optimization reduces the overhead by 15%.

Result: Anchor *is capable of resolving the conflicting interest between the participants of the resource market in the cloud. Its overall performance is competitive against that of optimization, with desirable properties of stability and VM-optimality of the outcome.*

D. Anchor at Scale

Finally, we evaluate the scalability of *Anchor* using both experiments and simulation. We conduct a medium-scale experiment involving all of our 20 machines. We orchestrate a complex scenario with 4 batches of allocation requests, each with 20 VMs of various sizes, followed by 2 batches of migration requests, each with 10 VMs of various sizes. Servers are initially empty with capacity of 7. VMs use distinct policies for both allocation and migration, and servers adopt a consolidation policy for allocation and an overhead-minimization policy for migration.

Since different agents have different objectives, we use the rank of the assigned partner in percentage as performance metric that reflects one's "happiness" about the matching. For servers, their happiness is the average rank of the matched VMs. From the experiment we find that VMs obtain their top 10% partner on average while servers only get their top 50% VMs. The reason is that the number of VMs is too small compared to servers' total capacity, and most of VMs' proposals can be directly accepted.

We then conduct large-scale simulations with the same setting. We vary the numbers of VMs and servers but keep their ratio constant, and evaluate the happiness statistics. The result is shown in Figure 18. On average VMs obtain their top 7% choice, while servers get their top 15-20% choices. Thus, we can conclude that *Anchor* remains effective in realizing clients' policies and resolving conflicting interest in large-scale problems.

We also plot the running time of Algorithm 2 in simulations to demonstrate the scalability of our algorithms. From Figure 19 we can see that the running time is less than 1 sec for problems of up to 8,000 VMs, and less than 8 sec when the problem size scales up to 12,000 VMs. Moreover, we observe that Algorithm 2 takes less than 4 stages to converge in most of the cases, and 7 stages maximum in simulations.

Result: Anchor *effectively realizes resource management policies in large-scale problems. Its multi-stage DA algorithm scales up to tens of thousands of VMs and takes less than 4 stages to converge in most cases.*

V. Related Work

Stable Matching. Since the seminal work of [11], a significant body of research in economics has examined different variants of stable matching, from one-to-one, many-to-one, to many-to-many models (see [26], [27] and references therein). Algorithmic aspects of stable matching have also been studied in computer science [16], [20]. However, the use of stable matching in networking is fairly limited. [18] uses a variant of the DA algorithm to map input to output queues in switching. Our recent work [34], [35] advocates stable matching as a general framework to solve networking problems. To our

knowledge, all these work assumed a traditional uni-size job model.

VM Allocation. VM allocation on a shared infrastructure has been extensively studied in the literature. [8], [28] presented methods to provision computation resources in data centers. [19] considered storage resources in addition and the resulted coupled placement problem. [24], [29] studied VM allocation from an energy perspective. Further, there are also studies to consider the correlation of network traffic between VMs and its effect on intelligent placement [21]. These work considered specific facets of the problem and proposed specifically crafted algorithms to deal with them. They are very complementary to *Anchor*, as the insights and strategies can be incorporated as policies to serve different purposes without the need to design another set of algorithms from ground up.

VM Migration. Besides efforts to enable efficient live migration of VM from an implementation perspective [9], [25], [31], there are also studies to develop automated strategies in determining which VM should be migrated to which servers [32]. In [14] automated VM migration is employed for scientific nano-technology workloads on federated grid environments. Shirako, a leasing contract based resource management system for a federated cluster environment, also used migrations to enable dynamic placement decisions [17].

VI. Concluding Remarks

We presented *Anchor* as a unifying fabric for resource management in clouds, where *policies* are decoupled from the management *mechanisms* by the stable matching framework. We demonstrated the use of stable matching as a better alternative to conventional optimization, by first developing a new theory of job-machine stable matching with size heterogeneous jobs as the underlying matching mechanism to resolve any conflict of interest between the cloud operator and cloud clients. We then showcased the versatility of the *preference* abstraction for a wide spectrum of resource management policies through a simple API. We described the design and implementation of the *Anchor* prototype system, and used a 20-node cluster testbed as well as large-scale simulations to evaluate its effectiveness, efficiency, flexibility, and scalability.

Our attempt of advocating stable matching in favor of optimization approaches here may seem ambitious. Yet, we believe that stable matching has unique merits, particularly its practicality, in solving problems in computer networking domain. It has potential, as a general framework, to offer fresh perspectives to the community. Towards this vision, we plan to explore *Anchor*'s applicability in other areas, such as resource management in federated computing, and job scheduling of MapReduce and Hadoop.

VII. Appendix

To prove Theorem 5 we need the following lemmas.

Lemma 3: If a particular job j participates in stage $t+1$, it must have participated in stage t.

Proof: It is a direct result of our algorithm design. When a job stops participating in $t+1$, it does not form any blocking pair with any of the machines no matter how the rest of jobs participated in t are matched. ■

Lemma 4: If a set of jobs do not participate in a particular stage t in Algorithm 2, then they are assigned to their best possible machine in all weakly stable matchings after t.

Proof: Let us call a machine "possible" for a given job if there is a weakly stable matching that assigns it there. We can prove this lemma by induction. It is clear that when we pick up all the jobs that can participate, it is equivalent to pick up those that definitely cannot participate. We can easily do so by first finding jobs that cannot participate unless some jobs that did not participate in previous stage participate, marking them, and finding the rest that cannot participate unless some jobs that were marked in this stage or previous stages participate, until there is no such job.

Assume that up to a certain point of Algorithm 2, the lemma holds, i.e. no marked job was permanently rejected by a possible machine (since it cannot propose any more). Now suppose we mark job j, which is (permanently) rejected by a possible machine m in a hypothetical matching μ'. Since j is marked, m must have accepted a set of jobs $\mathcal{J}_m$ that are preferable than j and marked before. Then, in μ', at least one of the jobs from $\mathcal{J}_m$ must be sent to a less desirable machine, since all machines preferable than m is impossible for it by assumption. This clearly forms a type-2 blocking pair in μ', and contradicts with the assumption. Thus the proof. ■

This shows that our algorithm only permanently rejects jobs from machines that are impossible to accept them in all weakly stable matchings, when the jobs cannot participate any further. The resulting assignment is therefore optimal.

Lemma 5: If $\mu_t = \mu_{t-1}$ in Algorithm 2, then the set of participating jobs at t are assigned their best possible machine in all weakly stable matchings.

Proof: Assume that at a given point of the algorithm execution at t, every job matched to its previous machine $\mu_{t-1}(j)$ is given its best possible machine. Suppose now, j is rejected by all machines better than $\mu_{t-1}(m)$, while there is a hypothetical weakly stable matching μ' that sends j to a better machine m. Thus j must have proposed to and been rejected by m. m rejected j because it accepted $j_1, j_2, \cdots$, each of which is preferable than j. If j_i did not participate in t, by Lemma 4 m is its best possible machine. If j_i participated in t, by assumption m is impossible for it. Thus in μ', for m to have j, at least one of j_i has to be sent to a less desirable machine, which causes a type-2 blocking pair (j_i, m) in μ', which contradicts the assumption. ■

Now we can prove Theorem 5.

of Theorem 5: There are two possible cases when Algorithm 2 terminates.

If the algorithm terminates when there is no type-1 blocking pair, i.e. no job can, or wishes to if it can, participate by proposing to a better machine. Then by Lemma 4, all these jobs are assigned to their best possible machine. Therefore the resulting matching is job-optimal.

If the algorithm terminates at stage t where $\mu_t = \mu_{t-1}$, then by Lemma 4 all jobs that did not participate in t are assigned the best machine. By Lemma 5 all jobs that participated in t are also sent to their best machine. Hence the matching is also the job-optimal weakly stable matching. ■

REFERENCES

[1] Apache Mahout, http://mahout.apache.org.

[2] Faban, http://www.opensparc.net/sunsource/faban/www, 2011.

[3] Olio, http://incubator.apache.org/olio.

[4] Oracle VirtualBox. http://www.virtualbox.org.

[5] VMware. http://www.vmware.com.

[6] Zenoss Inc., 2010 Virtualization and Cloud Computing Survey, 2010.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. SOSP*, 2003.

[8] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proc. OSDI*, 2001.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. NSDI*, 2005.

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

[11] D. Gale and L. S. Shapley. College Admissions and the Stability of Marriage. *Amer. Math. Mon.*, 69(1):9–14, 1962.

[12] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. ACM SIGCOMM*, 2009.

[14] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration. In *Proc. International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2006.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, 2009.

[16] R. Irving, P. Leather, and D. Gusfield. An Efficient Algorithm for the "Optimal" Stable Marriage. *J. ACM*, 34(3):532–543, 1987.

[17] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, and D. Becker. Sharing Networked Resources with Brokered Leases. In *Proc. USENIX ATC*, 2006.

[18] A. Kam and K.-Y. Siu. Linear Complexity Algorithms for Bandwidth Reservations and Delay Guarantees in Input-Queued Switches with No Speedup. In *Proc. ICNP*, 1998.

[19] M. Korupolu, A. Singh, and B. Bamba. Coupled Placement in Modern Data Centers. In *Proc. IPTPS*, 2009.

[20] D. F. Manlove, R. W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard Variants of Stable Marriage. *Elsevier Theoretical Computer Science*, 276:261–279, 2002.

[21] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *Proc. INFOCOM*, 2010.

[22] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Englightened Page Sharing. In *Proc. USENIX ATC*, 2009.

[23] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. ACM SIGCOMM*, 2009.

[24] R. Nathuji and K. Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proc. SOSP*, 2007.

[25] M. Nelson, B. H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proc. USENIX ATC*, 2005.

[26] A. E. Roth. Deferred Acceptance Algorithms: History, Theory, Practice, and Open Questions. *Int. J. Game Theory*, 36:537–569, 2008.

[27] A. E. Roth and M. Sotomayor. *Two-sided Matching: A Study in Game Theoretic Modeling and Analysis*. Number 18 in Econometric Society Monograph. Cambridge University Press, 1990.

[28] H. N. Van and F. D. Tran. Autonomic Virtual Resource Management for Service Hosting Platforms. In *Proc. IEEE CLOUD*, 2009.

[29] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server Workload Analysis for Power Minimization using Consolidation. In *Proc. USENIX ATC*, 2009.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proc. OSDI*, 2002.

[31] T. Wood, K. Ramakrishnan, P. Shenoy, and J. V. der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. VEE*, 2011.

[32] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. NSDI*, 2007.

[33] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *Proc. VEE*, 2009.

[34] H. Xu and B. Li. Egalitarian Stable Matching for VM Migration in Cloud Computing. In *Proc. IEEE INFOCOM Workshop on Cloud Computing*, 2011.

[35] H. Xu and B. Li. Seen As Stable Marriages. In *Proc. INFOCOM*, 2011.