Fair, Fast and Frugal Large-Scale Matchmaking for VM Placement

Nikolaos Korasidis[1], Ioannis Giannakopoulos[1], Katerina Doka[1], Dimitrios Tsoumakos[2], and Nectarios Koziris[1]

[1] Computing Systems Laboratory, National Technical University of Athens, Greece
{nkoras,ggian,katerina,nkoziris}@cslab.ece.ntua.gr
[2] Ionian University, Greece
dtsouma@ionio.gr

Abstract. VM placement, be it in public or private clouds, has a decisive impact on the provider's interest and the customer's needs alike, both of which may vary over time and circumstances. However, current resource management practices are either statically bound to specific policies or unilaterally favor the needs of Cloud operators. In this paper we argue for a flexible and democratic mechanism to map virtual to physical resources, trying to balance satisfaction on both sides of the involved stakeholders. To that end, VM placement is expressed as an Equitable Stable Matching Problem (ESMP), where each party's policy is translated to a preference list. A practical approximation for this NP-hard problem, modified accordingly to ensure efficiency and scalability, is applied to provide equitable matchings within a reasonable time frame. Our experimental evaluation shows that, requiring no more memory than what a high-end desktop PC provides and knowing no more than the top 20% of the agent's preference lists, our solution can efficiently resolve more than 90% of large-scale ESMP instances within $N\sqrt{N}$ rounds of matchmaking.

1 Introduction

VM placement is a fundamental problem addressed by current cloud providers [2, 1, 3]. The policy through which the VMs are placed into the physical hosts tremendously affects the data center's utilization, energy consumption and operational cost [1] while, at the same time, it greatly influences the VM performance and, hence, the cloud customer satisfaction. Albeit initially static and determined solely by the capacity and the utilization of the physical hosts, the VM placement schemes are becoming more sophisticated and democratic, taking into consideration the client's needs as well. Indeed, policies that increase both the data center's efficiency and the VMs' performance can prove a boon for providers and clients alike, keeping both sides content. For example, a user would benefit from collocating two VMs that require a low-latency network connection whereas the provider would also benefit from packing those VMs into a single host to save network bandwidth. In the general case, though, the interests of the two parties may also oppose each other, a fact that complicates decision-making.

Some works in the field examine the policy-based VM placement from a game-theoretic viewpoint, formulating the problem as a multi-agent game in which each agent tries to maximize her profit [9, 15]. The provided solution is a personalized pricing scheme, which mainly targets fairness among users themselves, rather than users and providers. Other approaches view the placement problem as a matching problem [14, 13, 11]. Instead of attempting to optimize a multi-dimensional utility function that encapsulates the objectives of both providers and users, each stakeholder retains a *preference list* that expresses her policy. The problem is then transformed into the Stable Marriage Problem (SMP) [7], where the objective is to identify a stable matching between VMs/VM slots.

Stability is a key concept in this formulation as it guarantees that there exists no pair of a VM (representing the cloud users) and a VM slot (representing the data center) that prefer each other more than their current match. Such a matching is proven to exist in any problem instance and it can be found by the Gale-Shapley algorithm [7]. Although the Gale-Shapley algorithm guarantees stability, it fails to ensure that the two sides are treated equally. On the contrary, due to the strict proposer-acceptor scheme it relies on, it inherently favors one side of the involved negotiators. Besides, finding an optimally fair stable matching is proven to be NP-hard [10]. Furthermore, the execution of the Gale-Shapley algorithm requires $\mathcal{O}(N^2)$ memory, N being the number of VMs, rendering it unsuitable for resource management in modern data centers that host hundreds of thousands of VMs.

To overcome the limitations of unfairness and quadratic memory growth, we propose a novel VM placement approach that seeks for a fair solution based on incomplete preference lists. Specifically, we generate preference lists based on the provider and user policies and apply the heuristic from our previous work in [8] to create a fair matching between the VMs and the available VM slots. To avoid quadratic memory expansion and scale to thousands of VMs, we reduce the preference lists to the top-K most preferable positions, rejecting any proposal originating from an agent lower on the list than the K^{th} one. Since this optimization modifies the problem's properties, we evaluate its impact on the algorithm execution and showcase that the usage of incomplete lists is suitable for most problem instances since equivalent matchings can be produced using only a portion of the original lists. Furthermore, through our extensive evaluation we study the parameters that affect the performance of our VM placement methodology and demonstrate that we can provide fair solutions even for problem instances up to $20k$ VMs within a few seconds, while using only 20% of the memory needed in the casual case without compromising neither the correctness nor the quality of the algorithm. Our concrete contributions can be, thus, summarized as follows:

- We formulate the policy-based VM placement problem as the sex-equal SMP and utilize an approximation algorithm [8] to identify a fair solution.
- We introduce a variation of the original approximation algorithm that relies on incomplete preference lists to be able to accommodate larger problem instances.

– We provide a thorough experimental study that evaluates our proposed method under various situations arising in modern data centers and demonstrates that our approach is particularly suitable for large-scale datasets where thousands of VMs must be scheduled for deployment.

2 Related Work

VM placement is a vividly researched problem in the cloud computing area. The suggested approaches can be categorized in two distinct classes: (a) the ones that aim to resolve the problem favoring exclusively the provider and (b) the ones that seek for a compromise between the provider's interest and the user-defined placement policies.

The first category includes approaches that optimize data center indicators such as utilization, operational cost, energy consumption, etc., while honoring the Service Level Agreements (SLAs). [2] focuses on the initial VM placement when a stream of deployment requests arrive to a data center. [1] proposes an energy efficient VM placement scheme that utilizes migration to relocate VMs so as to decrease the operational cost without jeopardizing the Quality of Service (QoS). In [4], a classification methodology is proposed with the objective to maximize the data center's efficiency, while the VM performance is kept at maximum. Similarly, in [12] the authors propose a discrete knapsack formulation, where the objective is to maximize a global utility function that corresponds to the operational cost with the constraint of respecting the SLAs. The authors utilize a Constraint Programming approach to tackle the problem. None of the above methods take into consideration user-defined scheduling policies.

Regarding the second category, we encounter two distinct problem formulations. The first, involves a game-theoretic approach, in which each customer is viewed as an agent that attempts to maximize utility. The objective of such formulations is to achieve an equilibrium in the game, so that the players are treated equally. For example, in [9], an approach is that attempts to provide personalized prices to each customer is presented. The idea is that each customer defines their workload and the provider needs to specify a price that is fair among the customers and is also beneficial for both the customer and the provider. Similarly, Xu et al. [15] present an approach that targets to provide fairness among the clients and also increase the cluster utilization.

Alternatively, the policy-based VM placement is formulated as an SMP, in which VMs are matched to the available hosts. This formulation entails the extraction of a preference list for each entity (i.e., VM and VM slot) that reflects the policies of the involved parties. [14] proposes a system that accepts user policies and executes the Gale-Shapley algorithm to find a mapping between VMs and physical resources while [11] uses the same algorithm to improve energy consumption. The resulting match in both cases, however, is proposer (i.e., Cloud provider)-optimal.

In our work, we utilize ESMA [8] to provide fair and stable solutions and propose an enhancement thereof to deal with the quadratic expansion of memory

requirements as the instance size increases. Equality in SMP is also considered in [13], where out of a number of different matchings constructed, the algorithm selects the one that maximizes fairness between the opposing groups. Contrarily, our approach produces a single fair solution, with performance comparable to the non-equal Gale-Shapley algorithm. This renders our work more suitable for real-time execution as new deployment requests arrive at the data center.

3 Preliminaries

An instance I of the *stable marriage problem* (SMP) consists of n men and n women, where each person has a *preference list* that strictly orders all members of the opposite gender. If a man m prefers w_1 to w_2, we write $w_1 \succ_m w_2$; a similar notation is applied to women's preferences. A *perfect matching* M on I is a set of disjoint man-woman pairs on I. When a man m and a woman w are matched to each other in M, we write $M(m) = w$ and $M(w) = m$. A man m and a woman w are said to form a *blocking pair* for M (or to *block* M) when: (i) $M(m) \neq w$; (ii) $w \succ_m M(m)$; and (iii) $m \succ_w M(w)$. A matching M is *unstable* if a blocking pair exists for M, and *stable* otherwise. The SMP calls for finding a stable matching M.

The standard algorithm for solving the SMP, proposed by Gale and Shapley [7], goes through a series of iterations. At each iteration men propose and women accept or reject proposals. The algorithm is guaranteed to terminate successfully after a quadratic number of steps, providing a perfect matching which, in effect, cater to the satisfaction of the proposers. Since many different stable matchings may exist for any problem instance, it is reasonable to aim for a matching that is not only stable, but also good by some quality metric.

3.1 Equitable Stable Marriage Problem

Past research has defined three quality criteria. Let $pr_m(w)$ (respectively, $pr_w(m)$) denote the position of woman w in man m's preference list (respectively, of m in w's list). The *regret cost* $r(M)$ of a stable matching M is:

$$r(M) = \max_{(m,w)\in M} \max\{pr_m(w), pr_w(m)\} \tag{1}$$

On the other hand, the *egalitarian cost* $c(M)$ is:

$$c(M) = \sum_{(m,w)\in M} pr_m(w) + \sum_{(m,w)\in M} pr_w(m) \tag{2}$$

Finally, the *sex equality cost* is defined as:

$$d(M) = \left| \sum_{(m,w)\in M} pr_m(w) - \sum_{(m,w)\in M} pr_w(m) \right| \tag{3}$$

A stable matching that optimizes the sex equality cost satisfies a notion of equity between the two sides of the involved stakeholders. Finding such a solution to the so-called *equitable stable marriage problem* (ESMP) is NP-hard [10], thus all proposed solutions use heuristics that either produce different stable matchings and seek for the fairest at the cost of increased execution time, or attempt to construct an equal approach by allowing proposers and acceptors to interchange their roles. The execution time penalty of the former category is prohibitive for our use case, since we need to be able to produce matchings in an online fashion. The Equitable Stable Matching Algorithm (ESMA) [8] is an algorithm of the latter category, that allows the opposite group members to both propose and accept in different algorithm steps. It is experimentally proven that ESMA, unlike its main competitor [6], (a) terminates on all tested large problem instances, (b) generates equitable matchings of high quality and (c) has an execution time similar to the Gale-Shapley algorithm, outperforming other similar solutions.

ESMA utilizes a simple non-periodic function in order to assign the proposer group. Each proposer proposes to their next preference and the acceptors evaluate their proposals and accept their most preferred choice rejecting the others, as in the classic Gale-Shapley algorithm. When the acceptors become proposers, they also start proposing to their most preferred choices only if the agents they propose to are most preferable than their current fiancé. Each agent retains two indices into their preference list: one that indicates their current fiancé and one that indicates their current most desired agent. These indices change positions while new proposals are issued and new marriages are established. The algorithm terminates when all agents of both groups have concluded that they are married to their most preferred choice that did not reject them, i.e., the two aforementioned indices point to the same agent.

As discussed in [8], this alternating proposing scheme may introduce repeating proposal patterns since two agents from the opposite groups may establish more than one engagements between them during the algorithm's execution, something which is not allowed in the Gale-Shapley algorithm due to the *monotonic* preference list traversal of the agents, i.e., the proposers start from the beginning of their lists and degrade as the algorithm evolves and the opposite stands from the acceptors. Since such a repeating pattern may compromise the algorithm's termination by creating endless proposal loops, the non-monotonic function utilized for assigning the proposer group seems to overcome this challenge and leads to termination for all the tested problem instances. In our work, we have not encountered any problem instance that lead ESMA to non-termination due to repeating cycles. This feature, in combination with the low per-step complexity ($\mathcal{O}(n)$) and the short execution times, establish ESMA as the basis for the VM placement scheme we propose at this work. Its application for VM placement is straightforward: We assume that one side consists of the clients' VMs to be deployed and the other side consists of the available VM slots. The final matching determines the slot that each VM will occupy.

3.2 Modeling preferences

The extraction of the preference lists from the policies defined by the user and the cloud provider is a tedious task, tightly coupled with the policy model. Various transformation schemes exist, depending on the nature of the policy. For example, [14] assumes that the users define their policies in a rule-based format, easily transformable to a serialized preference list for the available VM hosts. Specifically, the user generates *policy groups* in which the preference for a host is expressed in the form: $\texttt{1/CPU}_{freq}$ (the shorter the number the higher the preference), meaning that the higher CPU frequency hosts should be preferred. Composition of such simple rule-based policies would create a serialized preference list for each user and the provider, that also creates policies on a similar manner, based on data center metrics such as resource utilization, energy consumption, etc. On the contrary, in [9], a different policy scheme is adopted, where the user defines a graph that demonstrates the type of needed resources (e.g., cores, memory, etc.) and the timeline of their utilization. Obviously, such a policy description would entail a totally different preference list extraction algorithm. In this work, we mainly focus on what happens *after* the policy extraction. Since different schemes could be adopted in different use cases, we assume a simple policy mechanism like the one introduced in [14] and study various aspects of our approach when policies produce preference lists of different distributions and properties.

4 Dataset generator

To test our approach, we made use of synthetic data, i.e., preference lists generated using *Reservoir Sampling*. This is easy to implement and properly models the case where all agents are of the roughly same quality/fitness. However, in real life applications it is often the case that some portion of the resources available are globally more preferable to others. This variation in quality can be discreet (e.g., high quality agents VS low quality agents) or continuous. To incorporate this possibility, we assume that a weighting function exists that maps each agent to some *global bias*: agents with high global bias are more likely to end up in (the top places of) the preference lists of their prospective suitors. Thus, we implemented instance generation via *Weighted Reservoir Sampling*, following the A-Res interpretation [5]: If the sums of all weights/biases is B, an agent with bias b_i has probability b_i/B to appear in their suitors top preferences. Despite the biases, each agent's list is generated independently; randomness is provided via an agent-specific 32-bit *seed* which is derived from a global seed. Instance generation is entirely portable and reproducible, as it is based on PRNGs available in C++11.

Bias modeling happens through allocating different weights to different agents, and feeding those weights to WRS. We studied three different weight distributions, namely *uniform distribution*, *contested distribution* and *position-inverse distribution*.

In the uniform distribution (UNI) each agent is assigned the same, constant bias. This produces preference lists that are selected uniformly from all $K-$ permutations.

In the contested distribution (CONT), the agents are partitioned into two sets; one set comprises of higher-quality suitors that are globally preferrable, whereas the other set is medium-quality suitors. Agents in the high-quality set are assigned the same, constant weight that is 10 times larger than the weight assigned to agents in the lower-quality set. A parameter p controls the fraction of the agents that are considered high-quality. In our experiments we used $p = 0.1$.

Finally, in the position-inverse distribution (INV), each agent is assigned a weight that is a falling function of its id in the set, i.e. $i < j \implies b_{a_i} > b_{a_j}$. In our experiment, we used the function $f_a(i) = (i + 1)^{-a}$ to map integer ids to positive weights, for $a = 1.0$.

The reason for choosing these weight distributions is to model circumstances arising in the VM placement problem. Two parametres are of particular interest: the variation of a data center's equipment as time passes, and the variability in client's performance demands.

A new data center usually contains machines of about the same overall quality, which corresponds to UNI. Doing a minor hardware upgrade or expansion results in a small number of new machines becoming more powerful than the rest, a situation that corresponds to CONT. The cumulative effect of many such upgrades is a collection of machines with smoothly varying performance characteristics, corresponding to INV. The 2 latter ways of bias modeling result in preference lists that are not uniformly random $K-$ permutations: we expect the top fraction of those lists to be somewhat similar, creating a tougher problem instance.

Similarly, clients of a data center may be diversified in the quality of service they demand. The data center may provide only one level of QoS (modeled as UNI). Alternatively, it may provide two levels, an economical plan and a premium one (modeled as CONT). Finally, it may have some more refined service program, where the client pays more according to a smoothly varying parameter ("nines of uptime"), which we model with INV.

Out of the nine possible situations of biases, we have chosen those with the same bias on each side. This is done for simplicity but is not detached from the real world. A small scale data center would probably serve customers without special needs. On the contrary, any large data center serving a diversified clientele is probably old enough to have amassed equipment of varied performance.

5 Intuiting ESMP with trimmed lists

According to our formulation, achieving good VM placement reduces to solving large instances of ESMP. Unfortunately, the data set scales quadratically with N: an ESMP instance of $20k$ agents per set requires more than 6 GB of space. Solving larger instances requires resources usually available only in very high-

end machines. Thus, our primary goal is to drastically reduce ESMA's memory footprint.

5.1 An interesting metric

While running ESMA over randomly generated instances with at least 5000 agents, we are surprised to obtain solutions of low regret cost, despite never explicitly trying to optimize this cost. More importantly, the regret cost as a fraction of N *falls* as N grows larger, or, stated in plain words, as the pool of suitors expands, ESMA becomes increasingly effective at matching all agents to their top preferences. This low regret cost could just be a (welcome) artifact of the final matching, however, the statistics from our runs reveal a more systematic cause. It turns out that in any successful ESMA run all agents propose solely to their top preferences *throughout the matchmaking*. If we track the "proposer regret cost" over all rounds, we see that its maximum is only slightly larger than the final regret cost (the latter taken over all agents). So, although an agent acting as an acceptor may get proposed by a unfit suitor, no agent acting as a proposer does likewise. The following box-n-whisker plots are telltale.

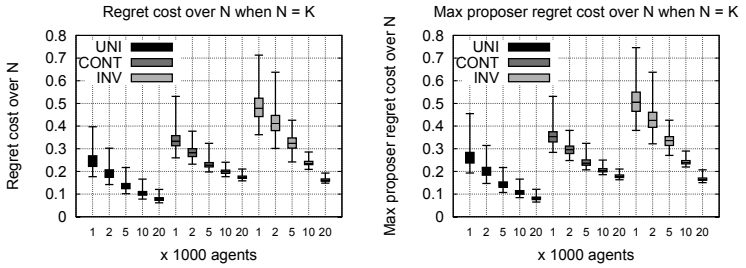

Fig. 1. Regret cost and max proposer regret cost for complete preference list

While proposers behave in a way that guarantees a low regret cost, acceptors may not necessarily do so: they may accept a low-ranking suitor as a partner. However, such an arrangement is not satisfying to them: For an agent to be satisfied according to ESMA, they must be matched to a partner that is within the fitness range they tolerate, which in turn is specified by the suitors they have been rejected from. As we noted already, an ESMA agent only proposes to (and hence is rejected from) suitors high in their preference list. Thus, for an agent to become satisfied when *accepting* a proposal, the proposer must be highly-preferred. An unsatisfactory match cannot be permanent. Therefore, when all agents are satisfied, it is because *all* are matched to a highly preferred partner.

5.2 Our approach

Intuitively, since no agent proposes to or forms a stable match with a lowly-preferred suitor, a fraction of the preference lists is effectively irrelevant to the solution: the ids of lowly-preferred suitors as well as the entries in the inverse map

that encodes the fitness of said suitors. We set out to investigate if it is possible to solve ESMP instances successfully while restraining ourselves to using only the topmost part of the preference lists, i.e. the part that encodes highly-preferred suitors. Our results, outlined in a following section, reveal that this is indeed the case; the majority of large instances can be successfully solved with at most 20% of the full data set, without sacrificing the quality of the solution.

5.3 Proof of stability

We can prove that whenever the algorithm terminates properly, the resulting matching is stable despite the incomplete lists. Consider an instance with N agents per set, each of whom has trimmed lists of length K. If $K = N$, we can rule out the existence of a blocking pair by invoking the stability proof cited in [8]: once all agents are satisfied, the resulting matching is stable and no agent can expect to be accepted by a more preferred suitor than their partner. In the case where $K < N$, we can show that no extra blocking pairs can form once the full lists are revealed, by reductio ad absurdum. Suppose such a pair (m_i, w_j) materialises only when they get to know their full lists, and that m_i prefers w_j to its current partner w_{m_i}. For m_i to sport such a preference only after the full lists are revealed, w_j must be among the trimmed preferences of m_i. But w_{m_i} must rank lower than w_j in his preference list, thus w_{m_i} also lies on the trimmed part, therefore m_i could never have engaged with her, a contradiction. Thus, no new blocking pairs form. Since all agents are satisfied, there were no blocking pairs initially. Combined, these two arguments prove that no blocking pairs exist at all; solving ESMP with trimmed lists produces stable matching given termination.

6 Experimental evaluation

Driven by the previous plots of low regret cost, we set out to experimentally investigate how much data can we ignore and still match agents successfully. Our testsuite consists of ESMP instances that vary according to three main parameters: number of agents per set N, fraction of preference list available K/N and bias modelling. We test all combinations of the following with 500 different seeds each:

- $N \in \{1000, 2000, 5000, 10000, 20000\}$
- $K/N \in \{0.1, 0.2, 0.5, 1.0\}$
- uniform bias, contested bias for $p = 0.1$, position inverse bias for $a = 1.0$

6.1 Ensuring correctness

We declare an execution to be `SUCCESSFUL` if it terminates properly with a matching within $N\sqrt{N}$ rounds of matchmaking, otherwise it is declared a `FAILURE`. The reason for choosing this particular round limit will be addressed in a later

section, along with an investigation of failed runs. In the following graphs, we plot the percentage of successful executions for each combination of parameters.

We observe that for any given fraction and bias tested, the percentage of successful executions increases with N or remains practically the same. This means that our approach is better suited for large, memory-demanding ESMP instances than for small ones. A most interesting finding is that the success rate initially increases as the lists get smaller but becomes zero after a threshold is reached, e.g., for 1k-5k agents in the Contested distribution case for $K/N = 20\%$. There appears to be a sweet spot for the list size, which depends on the problem size and weights used. We interpret this result as follows: As discussed earlier, no acceptor is stably matched to a suitor they prefer little. Reducing the list size outright prevents an acceptor from entertaining such a temporary match and forces the proposer to keep searching instead of idling. This reduces the number of rounds required until a globally stable match is reached, hence more executions terminate correctly within the given round limit. However, if the lists are trimmed too much, there are not enough options to accommodate everyone and hence the execution fails, as is expected in SMP with Incomplete Lists.

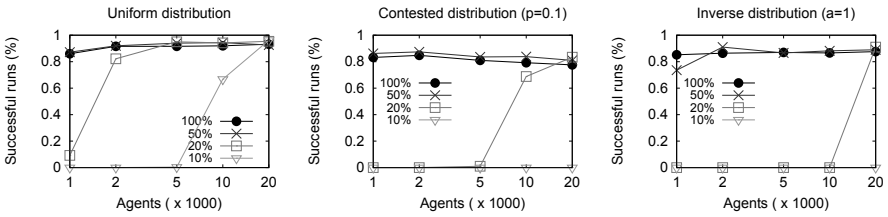

Fig. 2. Regret cost and max proposer regret cost for complete preference list

Other experiments, omitted due to space constraints, revealed that the "sweet spot" effect diminishes or even reverses completely if we allow for a very large round limit, e.g. in the order of $\mathcal{O}(N^2)$ In particular, all full-list instances terminate if given enough rounds, while some trimmed-lists instances seem to run forever. This phenomenon is interesting from a theoretical as well as a practical viewpoint.

As expected, uniform bias produces the easiest instances, position-inverse bias comes up second and contested bias is overall the hardest one. At $N = 20000$ and using only 20% of the full lists, we can solve more than 90% of instances of uniform and position-inverse bias, and more than 80% of instances produced via contested bias, using about 610 MB of RAM and at most about 2.8 million rounds. This demonstrates the power of our approach. In terms of execution, our approach needed approximately 15 seconds for the $N = 20k$, $K/N = 100\%$ case whereas for the same number of agents and smaller preference list (e.g.,$K/N = 10\%$) the algorithm only took 6.5 seconds. Both experiments were executed on a Linux machine using a 24 threaded Xeon X5650 CPU at 2.67GHz. The observed acceleration is attributed to the fact that shorter preference lists lead to faster access to the data structures which are allocated to support them. For more information, the reader can consult the Appendix A

6.2 Maintaining quality

Increased performance often comes at the price of quality. One would expect that the reduced amount of information causes our algorithm to select a worse matching. Fortunately, this effect appears to be quite mild, as detailed in the following plots. The regret and egalitarian costs remain virtually constant. The average sex-equal cost is negatively impacted, especially if N is small and we are not using uniform weights, but the difference is quite small. The variation of seq-equal cost increases the more we trim the lists. Again, the quality degradation is more intense for smaller datasets, something that indicates that our approach is more suitable for massive datasets, rather than smaller instances of ESMP which are easily solved by commodity machines.

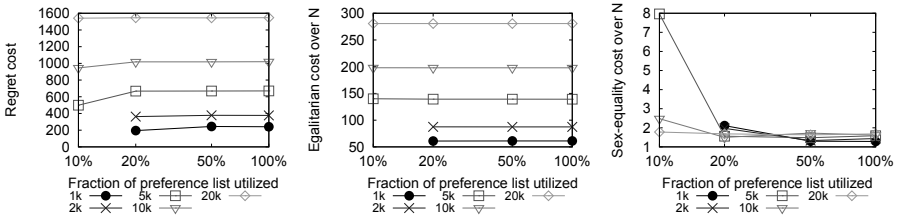

Fig. 3. Matching costs for different fractions of preference lists

6.3 Failing gracefully

As evidenced in the above graphs, no execution terminates successfully if preference lists are trimmed too much. This reduces the usefulness of our approach, as we cannot always predict the proper K for an instance a priori. We rectify this defect by showing that ESMA fails gracefully: it succeeds in matching the vast majority of agents to satisfying partners, even when a total matching is impossible. This happens even under extreme constraints: we run 500 ESMP instances with $N = 20000$ agents and lists trimmed at $K = 1000$, and recorded how many rounds passed until a sufficiently large fraction p of proposers are satisfied with their current partners, for $p \in \{90\%, 95\%, 99\%, 99.5\%, 99.9\%, 99.95\%, 99.99\%\}$. Our results appear below.

Table 1. Matched agents vs number of runs

Percentile matched	90	95	99	99.5	99.9	99.95	99.99
Gracefully failed runs	500	500	500	500	420	2	0

While none of the executions terminated successfully, most were able to come very close to complete stable matches in a surprisingly low number of rounds: 99.9% of proposers had been satisfied with their matches after about 30000 rounds, that is, within 1% of the allowed round limit. Since our instances used extremely small preference lists, it is reasonable to assume that a similar amount

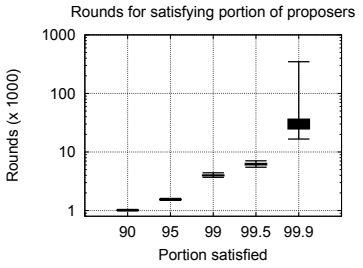

Fig. 4. Matched agents vs number of runs

of acceptors were also satisfied by the same number of rounds. Thus, after only few rounds of matchmaking, we can have a partial plan that places most VMs in appropriate slots and keeps most customers satisfied. The few resources that were not properly matched can be dealt with in an ad hoc way.

It should now be obvious that the arbitrary $N\sqrt{N}$ round limit was chosen out of pragmatic concerns. For large testcases, $\mathcal{O}(N^2)$ rounds of matchmaking (required by the GS algorithm to produce a complete stable matching in the worst case) is a prohibitively large amount given the setting of our problem. On the other hand, $\mathcal{O}(N)$ rounds seem to suffice for an almost complete partial match. We chose an intermediate value, so that our results can be used to generate useful guidelines. Depending on the specific workload and balancing the need for speed and completeness, a provider can choose their own round limit.

7 Conclusions

In this paper we revisited the problem of policy-based VM placement in a modern data center. In the equitable methodology we propose, an approximation algorithm was utilized in order to find a stable and fair matching between the VMs and the available VM slots, so as to honor the policies dictated by the opposite parties. Moreover, we developed an optimization of the original algorithm, in which we trimmed the preference lists of both groups to enable its scaling to hundreds of thousands of VMs, a typical case for modern data centers. Through an extensive experimental evaluation we showcase that our approach is able to find fair matchings for up to $20k$ VMs, which is an order of magnitude larger than other competitive approaches. Simultaneously, we demonstrate that trimming the preference lists is particularly effective for large cases, since both the correctness and the quality of ESMA's solutions, measured in terms of equality and global happiness, are maintained. As a future work, we seek confirmation of the results presented at this work with real-world workloads gathered in public data centers. Furthermore, we envision to port our methodology into streaming deployment requests so as to serve the needs of rapidly changing demands.

References

1. A. Beloglazov and R. Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, pages 826–831. IEEE Computer Society, 2010.
2. N. M. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti. Vm placement strategies for cloud scenarios. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 852–859. IEEE, 2012.
3. S. Chaisiri, B.-S. Lee, and D. Niyato. Optimal virtual machine placement across multiple cloud providers. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 103–110. IEEE, 2009.
4. C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.
5. P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.
6. P. Everaere, M. Morge, and G. Picard. Minimal concession strategy for reaching fair, optimal and stable marriages. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1319–1320. International Foundation for Autonomous Agents and Multiagent Systems, 2013.
7. D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
8. I. Giannakopoulos, P. Karras, D. Tsoumakos, K. Doka, and N. Koziris. An Equitable Solution to the Stable Marriage Problem. In *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on*, pages 989–996. IEEE, 2015.
9. V. Ishakian, R. Sweha, A. Bestavros, and J. Appavoo. Dynamic pricing for efficient workload colocation. 2011.
10. A. Kato. Complexity of the sex-equal stable marriage problem. *Japan Journal of Industrial and Applied Mathematics*, 10(1):1–19, 1993.
11. A. Kella and G. Belalem. VM Live Migration Algorithm Based on Stable Matching Model to Improve Energy Consumption and Quality of Service. In *CLOSER*, pages 118–128, 2014.
12. H. N. Van, F. D. Tran, and J.-M. Menaud. Sla-aware virtual resource management for cloud infrastructures. In *Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on*, volume 1, pages 357–362. IEEE, 2009.
13. H. Xu and B. Li. Egalitarian stable matching for VM migration in cloud computing. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 631–636. IEEE, 2011.
14. H. Xu and B. Li. Anchor: A versatile and efficient framework for resource management in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1066–1076, 2013.
15. X. Xu and H. Yu. A game theory approach to fair and efficient resource allocation in cloud computing. *Mathematical Problems in Engineering*, 2014, 2014.

Appendix

A Implementation details

Our implementation is a rewrite of the one presented in [8], with a few changes and additions to suit our purpose.

A.1 Languages & Frameworks

The original Java implementation was ported to C++11, as the later allows much greater precision in handling memory resources. We took care to make as much of the new code parallelizable and used the OpenMP framework to leverage any multicore power during instance generation, loading and solving.

A.2 Proposer selection

We propose a minor modification to the proposer-selection heuristic used in [8]: whenever all agents of one set are satisfied after some round, the other set is granted the proposer role in the next round. This ensures there will be some proposing action in all rounds save for the last one. It prevents idle rounds in large instances, and is especially impactful once only a few agents remain unsatisfied. Despite the tangible benefits, it is unclear whether this optimization may force the proposer-selection algorithm to fall in a periodic cycle, though none of our experiments indicate so.

A.3 Deterministic generation & subsetting

A problem instance generated by our framework can be reliably reproduced given the quintuple (N, n, K, b, s, h), where N is the number of agents per set, n is the size of the reservoir used, K is the length of the preference list, b is a function mapping agents to *positive* weights (floating-point values), s is an initial seed (64-bit integer) and h is a hash function. We use the 64-bit Mersenne Twister (MT64) engine available in the Standard Library of C++11, a fast PRNG that guarantees long periods.

We generate a preference list as follows. First, we hash the global seed s together with the agent id to create an agent-specific seed s', with which we setup a MT64. We feed the engine and the function b together with N and n to WRS and obtain a sample of suitors of length n, where $n \leq N$. Each sampled suitor is paired with the weight which put them in the reservoir. As a last step, we extract the heaviest K suitors (where $K \leq n$) in the sample and place their ids in the preference list ordered according to their weight. Each agent's list can be generated independently; our OpenMP implementation takes advantage of this.

Consider two instances occurring from the same (N, n, b, s, h) but differring in the cut-off point: $K_1 < K_2 \leq n$. The sampling phase is independent from K

and so will produce the same sample of length n for any two agents with the same id. Since the suitors are placed in the preference list according to the order of their weights, the list of an agent in the K_1 instance will coincide with the first K_1 elements of the corresponding agent's (longer) list in the K_2 instance. This effect enables us to generate instances that are *proper subsets* of other instances. We can thus study with great precision the effects of limiting the preference list at any desired fraction.

A.4 Compact representation of inverse map

The bulk of each agent's dataset consists of two structures: the preference list, which contains the suitors in order of preference, and the inverse map, which maps a suitor's id to their index in the agent's preference list. When entire preference lists are used, the inverse map can be represented as a plain array. However, once the preference lists are trimmed, the array becomes sparse and it is wise to represent it as a static dictionary in order to save space. We explored several concrete implementations built around `stl::unordered_map` and `boost::flat_map`. The former one is very fast but the latter one achieves a much better balance between performance and memory consumption. For our experiments, we re-implemented `boost::flat_map` from scratch to reduce unecessary overheads and augmented it with a summarizer cache that enables faster searches.