## Appendix A
## More discussion on the atomic VM sizing assumption

Readers may be concerned that different jobs will have varying resource requirements that cannot be accommodated by the simple atomic size assumption. For example, hadoop and data analytics jobs have stringent requirement on CPU and bisection bandwidth, while database applications require more I/O and memory. We notice that in reality, public clouds have some special VM configurations that do not follow the atomic sizing assumption for these jobs. For example Amazon EC2 provides high-memory, high-CPU, high I/O, and cluster compute instances specialized for certain applications. It is highly likely that they are managed separately with a different infrastructure. Measurement results [1] corroborate our arguments while public information is yet to be made available on the specifics of the Amazon infrastructure. We thus take the liberty to adopt this assumption, and believe that this represents a close resemblance of reality while keeping the problem analytically tractable[1].

## Appendix B
## Additional examples for Sec. 3 and 4

### B.1 An example where the classical DA algorithm may fail for job-machine stable matching

Fig. 1 shows an example similar to Fig. 2. Say we first let jobs $a, b, c$ propose until they cannot. The result is $A-(c), B-(b)$, 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 $B$.

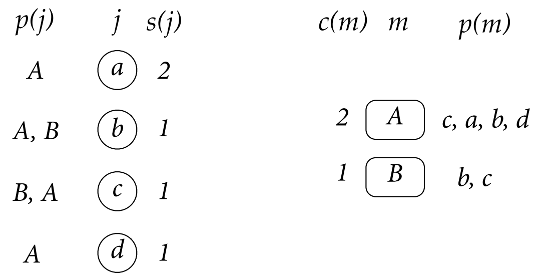

Fig. 1. An example where a possible execution of the DA algorithm produces a type-2 blocking pair $(b, A)$.

On the other hand, if we let $d$ propose to $A$ first before $a$ and $b$, and keep the rest of the execution sequence unchanged, the result becomes $A-(c), B-(b)$, which is weakly stable. This example demonstrates two problems when applying the classical DA algorithm here. 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 a weakly stable matching for an arbitrary problem.

1. Without the atomic sizing or any other assumption to reduce the dimensionality of resources, the problem is essentially a multiple knapsack problem which is known to be NP-hard [2]. Multi-resource allocation is an open problem for which there has been some progress made recently in understanding the problem [3].

### B.2 An example where running `Revised DA` multiple times may result in an unstable matching

To give an example, let us take a look at the problem in Fig. 2. We now run `Revised DA` 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.

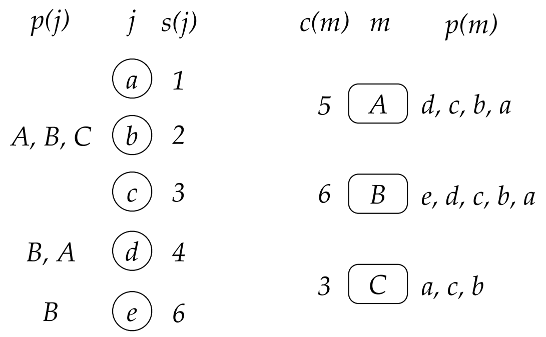

Fig. 2. An example where simply running the revised DA algorithm multiple times will produce a new type-2 blocking pair $(c, C)$.

## Appendix C
## Additional theoretical results

### C.1 Proof of Lemma 1

*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. □

### C.2 Proof of Theorem 7

To prove Theorem 7 we need the following lemmas.

***Lemma* C.1:** 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* C.2:** If a set of jobs do not participate in a particular stage $t$ in `Multi-stage DA`, 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 the 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 `Multi-stage DA`, 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 $\mu'$. 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 $\mu'$, 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 $\mu'$, 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* C.3:** If $\mu_t = \mu_{t-1}$ in `Multi-stage DA`, 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 $\mu'$ 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 C.2 $m$ is its best possible machine. If $j_i$ participated in $t$, by assumption $m$ is impossible for it. Thus in $\mu'$, 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 $\mu'$, which contradicts the assumption. □

Now we can prove Theorem 7.

*Proof of Theorem 7:* There are two possible cases when `Multi-stage DA` 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 C.2, 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 C.2 all jobs that did not participate in $t$ are assigned the best machine. By Lemma C.3 all jobs that participated in $t$ are also sent to their best machine. Hence the matching is also the job-optimal weakly stable matching. □

### C.3 Proof of Theorem 8

*Proof:* This can be easily proved by contradiction. Assume that the matching produced when the algorithm terminates with no job proposing is not strongly stable. Thus, we must be able to find a type-1 blocking pair, say $(j, m)$, as implied by Theorem 5. $m$ 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 left for future work.

## APPENDIX D
## AN ONLINE ALGORITHM `Online DA`

### D.1 Algorithm design

In this section, we address the technical challenge of devising an online algorithm. We extend the static problem setting to a dynamic environment, where the set of jobs $\mathcal{J}$ arrive *online* instead of offline[2].

We observe that our `Revised DA` algorithm can be readily used in the online setting. The high-level idea is that, we fix the previous matching and improve upon it by applying `Revised DA` only on the set of new jobs. The reason to fix the previous matching is to avoid the potential overhead of and downtime caused by VM migration that may occur if we allow the previous jobs to participate with the new jobs. Since most resource management policies for clients, as we shown in Sec. 5.2, depend on system state variables that change during the placement process, the preferences of new VMs need to be configured according to the most updated state variables.

Let us now present the details of the online version of `Revised DA` called `Online DA` as shown in Table 1. First of all, the execution never stops. After a matching is found, it waits for new requests. When a new set of jobs $\mathcal{J}'$ arrive, their preferences are configured according to the current system state variables. The server preferences

2. Presumably, the set of machines is fixed for a long period of time.

are configured with regards to the new jobs only. Then the Revised DA algorithm runs with the new jobs $\mathcal{J}'$ and all the servers $\mathcal{M}$. Note that in step 7 $c(m)$ is the current server capacity, since it is always updated during the running of Revised DA.

TABLE 1
Online DA

```
1: Input: $c(m), \forall m \in \mathcal{M}$
2: Initialize all $m \in \mathcal{M}$ to free
3: while true do
4:     Wait until new jobs $\mathcal{J}'$ arrive
5:     Configure preferences of new jobs $j' \in \mathcal{J}'$ $p(j)$ according
       to the current system state.
6:     Configure preferences of all servers $p(m)$ with regards to
       new jobs $\mathcal{J}'$.
7:     Run Revised DA with $c(m), p(m), \forall m \in \mathcal{M}$,
       $s(j), p(j), \forall j \in \mathcal{J}'$.
8:     Output the current matching
9: end while
```

## D.2 Discussions

Readers may wonder if our Multi-stage DA algorithm also can be revised to be an online algorithm. We choose not to use Multi-stage DA because it is iterative and takes much longer to converge when the problem scale is large, which may not be acceptable in practice. This is experimentally demonstrated in Sec. 6.3. Thus Multi-stage DA is only to be used as an offline algorithm for small to medium scale static problems, while the simple Revised DA and Online DA can be used in offline and online fashion, for even large-scale problems. We thus consider Revised DA and Online DA more suited for practical use of large-scale dynamic cloud systems, while Multi-stage DA more of theoretical value in exploring the intricacy of the job-machine matching problem with size heterogeneity.

## APPENDIX E
## ADDITIONAL POLICY EXAMPLES

The versatility of the preference concept can be further shown in the following examples.

**Colocation/anti-colocation.** In practice, some clients do or do not want their VMs to be placed with VMs of some other tenants. *Anchor* supports this policy using the API call colocate(tenants, i, *g_c*), where tenants is a list of tenants, i is a boolean variable to indicate whether the caller wishes to colocate or anti-colocate itself with clients in tenants, and *g_c* is an optional argument indicating the common policy group the colocating tenants belong to.

In case a client $A$ wishes to colocate with a set of clients $\mathcal{B}$, it makes a call to colocate($\mathcal{B}$, true, g). Since colocation is a mutual agreement, client $A$ and clients in $\mathcal{B}$ must share a common policy group g which is assumed to be agreed upon beforehand. The provider then bundles client $A$'s VM with VMs of clients in $\mathcal{B}$ as *virtual* VMs, where each virtual VM consists of an individual VM of $A$ and an individual VM of each client in $\mathcal{B}$. Such a virtual VM is then treated as a single VM to join the policy group g and participate in Anchor's stable matching algorithms.

In case a client $A$ wishes to anti-colocate with a set of clients $\mathcal{B}$, it can indicate so with colocate($\mathcal{B}$, false). The provider partitions the entire set of servers into two non-overlapping sets $\mathcal{X}$ and $\mathcal{Y}$. It creates for $A$ a new policy group g_A containing servers in $\mathcal{X}$ only, and for each client in $\mathcal{B}$ a new policy group g_b containing servers in $\mathcal{Y}$ only. Client $A$ and clients in $\mathcal{B}$ can then configure their own policy groups for their preferences. It is guaranteed that client $A$ and clients in $\mathcal{B}$ propose to non-overlapping sets of servers and will not be colocated. Note that anti-colocation is not a mutual agreement, and thus clients involved can have distinct preferences and policy groups.

**Tiered service.** It is a common practice to implement tiered services in an operational cloud, by associating each VM with a priority class. This can be done in *Anchor* with a call to conf(g_o, priority).

**Incomplete preferences.** It is possible that some VMs can only be placed on a subset of servers, due to hardware constraints for instance. Such placement limitations can be accommodated in *Anchor*, as the list of preference does not need to contain the entire set of servers. The policy engine in *Anchor* supports an additional API call limit(g_c, servers) so that VM preferences in a client policy group g_c only include servers specified in servers.

**Combination of policies.** Besides individual policies, *Anchor* also supports combinations of policies, which is also common in practice. The API call conf(g, factor1, factor2, ...) is designed for this purpose, where different policies are input following a decreasing priority order. A multi-pass sorting procedure is then performed, first based on the least important factor, then the second least, and so on, to produce preferences that adhere to this combination of policies.

## E.1 Discussion

Most preference examples shown above for clients are based on state variables that change during the placement process. However, we emphasize that the preferences do not change during the placement process, for otherwise the concept of stability cannot be well defined with respect to the constantly changing preferences. Our algorithms thus produce a stable matching with respect to the preferences and system state given before the placement process. This results in sub-optimal client performance compared to other algorithms that keep track of state variables during the process. This is an inherent limitation of the stable matching framework when applied to cloud resource management, because many practical policies rely critically on system state variables. We believe that it is an important and difficult problem, and would like to study it further as our future work.

## Appendix F
## Additional evaluation results

### F.1 Implementation details

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 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 a placement request arrives, it obtains necessary information from the resources monitor's database, and sends sorted preferences to the matching engine.

The matching engine implements the `Revised DA` and `Multi-stage DA` in the offline case, as well as the `Online DA` in the online case in Python. We preprocess server preferences (with a complexity of $O(n)$) to obtain an inverse of the list indexed by VM ID. 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.

### F.2 Small-scale experimental evaluation of *Anchor*'s effectiveness

This experiment uses both consolidation and load balancing policies for VM placement to demonstrate the effectiveness of *Anchor* in realizing resource management policies. We assume that clients follow the operator's default policy in the experiment here, so there is no conflicting interest involved.

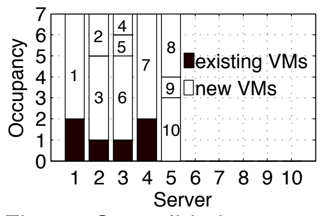

Fig. 3. Consolidation.

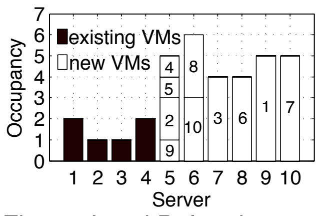

Fig. 4. Load Balancing.

The experiment is to allocate 10 VMs to 10 servers, the first four of which are configured with an occupancy of 2, 1, 1, 2, respectively. Fig. 3 shows the result of using the consolidation policy, where VM preference is ranked in descending order of server occupancy, 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 occupancy is thus maximized. On the other hand, the load balancing policy distributes VMs across the idle servers, resulting in a more balanced server load as shown in Fig. 4. `Multi-stage DA` takes 3 iterations to converge to the strongly stable matching of Fig. 3 for the former case, and 2 iterations for the latter.

### F.3 Evaluation results on `Online DA`

We conduct large-scale trace-driven simulations to evaluate *Anchor* in a dynamic environment where VM placement requests arrive dynamically. We again use the RICC workload trace as the input to our `Online DA` algorithm. The trace contains each task's arrival time, finish time, and amount of resources requested. In our simulation, we process each task scheduling request in the trace according to its arrival time as VM placement request(s) of various sizes, and use `Online DA` to determine a matching for them. When the task finishes, we remove its VMs from the corresponding servers, so capacity is freed to accommodate future requests.

In our simulations, the servers are configured to use the consolidation policy. The VMs are again randomly chosen to join the default, the CPU-bound, and the memory-bound policy group that uses the consolidation policy, the CPU bound and the memory-bound resource hunting policy, respectively. The free CPU and memory of a server is simply calculated as the total CPU and memory minus the amount allocated to its active VMs. We emphasize that VM preferences in this case are configured upon arrival using the most current system state variables as discussed above. Large tasks that require more than one server is translated into multiple smaller requests.

We simulate the exact birth-death history of the first 2000 tasks in the RICC trace, amounting to 16600 VM placement requests that cover a time period of 475529 seconds, or roughly 5.5 days. We compare `Online DA` with the online version of the first fit algorithm that first sorts the VMs according to their size (recall servers use a consolidation policy), and then tries to place a VM to the best machine according to its preference that has sufficient capacity to take it upon its arrival. The number of servers is fixed at 1024.

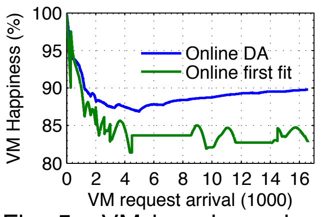

Fig. 5. VM happiness in a dynamic setting.

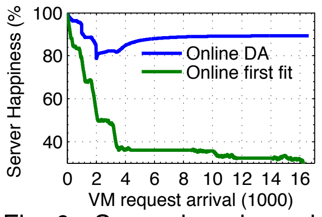

Fig. 6. Server happiness in a dynamic setting.

The results are shown in Fig. 5 and 6. We can see that in terms of VM happiness, `Online DA` only slightly outperforms `Online first fit` by about 7%. However in terms of server happiness, `Online DA` enjoys a significant improvement where servers obtain their top 1% of the VMs while `Online first fit` is only able to match servers with their top 60% VMs.

The reason of the performance discrepancy is again due to the elegant propose-reject design of stable matching algorithms. As discussed in Sec. 6.3, the first fit algorithm will not match a VM to a server whose capacity is insufficient, while `Online DA` will if this VM is preferable than some of the server's VMs during its execution. This improves the happiness of both VMs and servers, since VMs can get better machines even when they are occupied, and servers can also get their favorable VMs.

We also evaluate the complexity of `Online DA`. Fig. 7 shows the running time of `Online DA`, including processing preferences, for the entire course of simulation. Observe that it takes `Online DA` less than 3 seconds in any case to find a matching for new VMs, making it responsive to dynamic situations. Fig. 8 further shows that the algorithm usually terminates in less than 50 iterations.

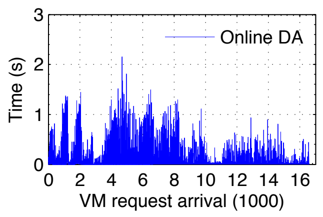

Fig. 7. Running times of `Online DA`.

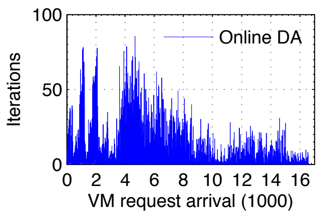

Fig. 8. Number of iterations of `Online DA`.

*Result: The* `Online DA` *algorithm is responsive and efficient to handle dynamic VM placement requests. Its performance is significantly better than existing methods based on the first fit algorithm.*

### F.4 Deployment in Real Cloud Environment

Here we discuss some potential challenges related to the real-world deployment of *Anchor* in a cloud environment.

The virtualized cloud infrastructure does not need any change to deploy *Anchor*. All the major virtualization solutions, including VMware, Xen, KVM, etc., collect resource utilization statistics through the hypervisor and provide API to access the information. *Anchor* can directly utilize the hypervisor APIs for its resource monitor. The cloud's client API needs to be augmented to include *Anchor*'s policy interface as shown in Table 3 of the main file for clients and the operator to express their policy goals. Given that the cloud operator has total control over its own client API and interface, this is not difficult to implement in reality.

In terms of scalability, *Anchor*'s *Revised DA* and *Online DA* algorithms are demonstrated to be efficient in handling large-scale VM placement problems, while *Multi-stage DA* may only be suitable for small-scale problems in Sec. 6. For large-scale problems, the possible limitation is from the network interconnect, which may become the bottleneck for transmitting the bulky OS image files for initializing VMs. This has started to gain attention in some recent work [4].

## Appendix G Related work on job scheduling

There have been tremendous efforts on job scheduling in a computer cluster or grid. [5], [6] represent several early works that focus on minimizing the expected completion time of all jobs by strategically choosing their execution sequence. [7], [8] discuss the online algorithm design problem for grid computing. The focus is on the completion times of jobs, which in our case are not known a prior to the scheduler of a cloud since users may use VMs for as long as they wish. For more complete coverage of the literature readers are directed to [9], [10]. [11] present some recent efforts of developing scheduling algorithms taking into account the practical constraints of a grid, including the trade-off between cost and time in a grid resource market, the QoS, etc., while [12] analyzes the performance of datacenters using queueing theory.

The distinction between this line of work and ours is clear: in cloud computing with VMs, the major difficulty of job scheduling comes from the inability to unify the different factors that operators and clients may consider as a single utility function, and from the packing constraint represented by the size heterogeneity of jobs. Our use of stable matching theory resolves the problem by adopting the stability concept, and new algorithms are developed to specifically tackle the size heterogeneity of jobs.

## References

[1] G. Wang and T. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proc. IEEE INFOCOM*, 2010.

[2] M. Korupolu, A. Singh, and B. Bamba, "Coupled placement in modern data centers," in *Proc. IPTPS*, 2009.

[3] C. Joe-Wang, S. Sen, T. Lan, and M. Chiang, "Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework," in *Proc. IEEE INFOCOM*, 2012.

[4] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *Proc. IEEE INFOCOM*, 2012.

[5] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, no. 2, pp. 280–289, April 1977.

[6] S. K. Sahni, "Algorithms for scheduling independent tasks," *J. ACM*, vol. 23, no. 1, pp. 116–127, January 1976.

[7] W. Leinberger, G. Karypis, and V. Kumar, "Job scheduling in the presence of multiple resource requirements," in *Proc. ACM/IEEE Conference on Supercomputing*, 1999.

[8] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke, "Online scheduling to minimize average stretch," in *Proc. IEEE FOCS*, 1999.

[9] V. K. Naik, M. S. Squillante, and S. K. Setia, “Performance analysis of job scheduling policies in parallel supercomputing environments,” in *Proc. ACM/IEEE Conference on Supercomputing*, 1993.

[10] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *Elsevier Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, June 2001.

[11] M. A. Salehi, B. Javadi, and R. Buyya, “QoS and preemption aware scheduling in federated and virtualized grid computing environments,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 72, no. 2, pp. 231–245, 2012.

[12] H. Khazaei, J. Mišić, and V. Mišić, “Performance analysis of cloud computing centers using M/G/m/m + r queueing systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, 2012, to appear.