Thinner Clouds with Preallocation*

Ittay Eyal[1], Flavio Junqueira[2], and Idit Keidar[1]

[1]Department of Electrical Engineering, Technion, Haifa, Israel
[2]Microsoft Research, Cambridge, UK

Abstract

Different companies sharing the same cloud infrastructure often prefer to run their virtual machines (VMs) in isolation, i.e., one VM per physical machine (PM) core, due to security and efficiency concerns. To accommodate load spikes, e.g., those caused by flash-crowds, each service is allocated more machines than necessary for its instantaneous load. However, flash-crowds of different hosted services are not correlated, so at any given time, only a subset of the machines are used.

We present here the concept of *preallocation* — having a single physical machine ready to quickly run one of a few possible VMs, without ever running more than one at a given time. The preallocated VMs are initialized and then paused by the hypervisor. We suggest a greedy preallocation strategy, and evaluate it by simulation, using workloads based on previous analyses of flash-crowds. We observe a reduction of 35-50% in number of PMs used compared with classical dynamic allocation. This means that a datacenter can provide per-service isolation with 35%-50% fewer PMs.

1 Introduction

Online applications increasingly use cloud providers to host production services. Such applications, from small start-ups to large web sites, delegate the physical management of computer resources, and host their services with an external cloud provider. When using cloud infrastructure, companies typically follow the pay-as-you-go model, paying only for the resources they use at a given time. They run a small number of machines most of the time, and increase this number when they wish to accommodate a larger workload.

Applications are typically comprised of a number of components, such as front-end servers, business logic, and storage. Every service in a cloud is provided by a number of *virtual machines*, or *VMs* for short, at any given time. Each VM runs on a *physical machine*, or *PM*, which is a cloud server running a hypervisor. A load balancer routes incoming workload to the services, distributing it homogeneously among each service's VMs.

Due to the cost of ownership of a PM, which might be as high as double the price tag of the server itself, it is desirable to provide a given set of services using as few PMs as possible. This is precisely the goal of this work. Note that we do not deal with the orthogonal problem of saving energy by putting PMs to sleep, but rather with reducing the number of PMs in the data center.

We consider elastic cloud-hosted services that should quickly scale when required, *e.g.*, due to flash-crowds. These include front-end or business logic tiers that do not store long term session state on behalf of clients. Such services are elastic in nature, as machines can be added or remove according to load machines. However, to run a VM on a PM, the image of the VM has to be loaded onto the PM and bootstrapped. As this process is time consuming [4, 20], it has to be done in advance to accommodate load spikes.

In principle, it is possible to consolidate, and run multiple VMs on the same PM. This would allow a single PM to serve varying loads with its running VMs. However, despite progress in performance isolation, current techniques still present drawbacks [12, 17, 7], and cloud users are generally hesitant to have multiple production VMs share a core. Additionally, for multi-tenancy, in which case a VM of one user may be co-located with that of another, the danger of side-channel attacks [18, 19] is another significant reason to avoid VM core sharing. Our working hypothesis is therefore that multiple VMs do not concurrently run on the same PM.

This paper addresses the challenge of decreasing the number of PMs without violating service isolation. We

*Work conducted during an internship at Yahoo! Research. This work was partially supported by the Hasso-Plattner Institute, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), the Technion Autonomous Systems Program (TASP) and the Israeli Science Foundation (ISF). The authors thank Edward Bortnikov for his good advice.

present a novel approach for managing VMs in a cloud provider that consists of *preallocating* multiple VMs on each PM (Section 3) and show that it leads to a substantial reduction on the number of PMs a cloud provider needs to host. With our approach, all preallocated VMs are loaded and bootstrapped, and then put to standby. Note that preallocated VMs consume neither CPU time nor network bandwidth, though they do occupy disk and memory space. A PM can run any single one of its preallocated VMs at any given time. This way, no security or performance isolation concerns arise.[1]

Our approach raises the question of choosing a *preallocation strategy*, *i.e.*, which VMs to preallocate in each PM; we tackle this question using a simple greedy heuristic.

We evaluate the effectiveness of preallocation in Section 4. Our workloads are based on analysis of flash-crowd behavior in the Internet [3, 21, 8]. Even with our simple heuristic, we observe a reduction of over 35% and 50% with preallocation of 2 and 3 VMs per PM, respectively. This means that with preallocation of as little as 2 VMs per PM, we need a datacenter 35% smaller, running an average of 35% fewer machines at any given time.

We note that despite its resemblance to bin-packing, optimal routing, and variants thereof, we have not found previous work that directly applies to the problem of choosing a preallocation in the literature. Related work is discussed in Section 6.

Our work is only a first step towards exploiting preallocation in datacenters. We present here only our simple preallocation strategy, and more elaborate schemes may improve the benefits achieved with preallocation even further. Using machine-learning tools that estimate the workload can allow more fine-tuned preallocation decisions. Finally, experimenting with real workloads in a real production system is necessary to prove the feasibility of our approach. Section 7 concludes the paper with a discussion of future directions.

2 Datacenter Architecture

A data center contains a set of physical machines, each running a hypervisor. Multicore physical machines, commonly found in data centers, use one core for the hypervisor, and the others are available for running one VM each. To simplify presentation, we assume that each PM has one core available for running a single VM.

The data center hosts a set of services. Each service is implemented with a set of stateless VMs running identical images. Clients use a service by issuing requests against it. An example of a request is a search query. A request is self contained, and any VM of the service can process it. The workload of a service is evenly distributed across the service VMs using a load-balancer.

We measure workload in units of VMs required to run them. For example, a workload of 3.5 requires 4 VMs running on 4 PMs. If the workload of a service exceeds the number of running service VMs, we say that there is an *overflow* that is given by the number of missing machines. For example, a workload of 3.5 of a service running 2 VMs at a given time has an overflow of 1.5. An overflow implies that the available machines are loaded above their preferred work point, and may not behave as they should.

The system is managed by an entity called the *orchestrator*, which is in charge of loading the VMs on the PMs, instructing the hypervisors to run the preloaded VMs, receiving statistics from the load balancer, and updating it on which VMs are available for each service.

The orchestrator typically allocates one VM per PM, and this PM is then able to run the allocated VM. To accommodate workload spikes, each service has more VMs allocated than necessary for the average workload. The service is then ready to handle occasional spikes without impacting the quality of the service offered. As the workload changes, the orchestrator can deallocate a VM, and allocate another one in its slot. Allocation includes loading the VM's image and bootstrapping it — a lengthy process taking often several minutes [4, 20].

Slack-factor allocation

With slack-factor preallocation, we assume that the load of the different services is independent and similarly distributed. First, we choose a slack-factor, α. The slack factor is one plus the fraction of spare resources we make available to the service. Then, for each service i at time t with load w_i^t, the orchestrator tries to allocate the service's VM in $\alpha \times w$ physical machines. For example, if $\alpha = 1.2$, then we allocate 20% more physical machines than currently used. If not enough VMs are allocated at t, it starts allocating on new machines as necessary. A larger slack factor therefore increases the number of PMs used, and decreases the probability of overflow.

3 Preallocation

We now describe our technique for reducing the number of PMs required for a system by allocating multiple VM images on each PM, and switching between them dynamically. Preallocation is an extension of allocation. With preallocation, a PM is able to run more than one service in less time than that of a full allocation.

In this work, we take the following simple approach. A preallocation includes loading the VM's image, bootstrapping it, and pausing it using the pause (also known as suspend) mechanism of the hypervisor [1, 2]. A preallocated VM occupies both the disk and RAM of the PM,

[1]Preventing malicious VMs from gaining control over their hosting hypervisor is outside the scope of this work. Preventing such attacks is essential for cloud computing with or without preallocation.

but it is swapped out, and does not receive CPU cycles or network bandwidth. To preserve RAM space, much of a suspended VM's memory could be swapped out, requiring a short warm-up period when it is swapped back in; we ignore this latency, which is small compared to the other time constants.

This scheme prevents all the concerns raised due to the fact that VMs run on the same PM, since context switches are infrequent, and effectively a PM runs only one VM at a time. However, it is possible to switch the running VM almost instantly, so when the workload of a certain service spikes, it is possible to quickly run the appropriate VMs, if they were preallocated on PMs that are not needed for other workloads at the same time.

3.1 Operation

Like classical dynamic allocation, the orchestrator allocates VMs on PMs according to need. This includes loading the VM image to the PM, bootstrapping it and running the VM. Unlike standard dynamic allocation, however, the orchestrator may suspend the running VM, and allocate an additional VM on the same PM. The number of VMs on each PM is bounded by disk and memory capacity. The maximal possible number of VMs that may be preallocated on a PM is called the *preallocation factor*. For a preallocation factor of x, we call it x-preallocation. Upon a command from the orchestrator, a PM can swiftly switch between the different VMs allocated on it. If necessary, the orchestrator may deallocate a preallocated VM, clearing its slot for a different VM to be allocated. Classical dynamic allocation is therefore preallocation with a factor of one.

As the workload changes, the orchestrator preallocates VMs and chooses which VM to run on each PM. It updates the load balancer accordingly. The exact strategy a load balancer should use is out of scope.

In summary, the orchestrator performs the following operations:

- Allocate a VM on a PM if there are less VMs than the preallocation factor allocated on the PM.
- Deallocate a VM from a PM to free a slot.
- Shut down a PM (equivalent to deallocating all its allocated VMs).
- Start a new PM, and start allocating a VM on it.
- Start running a VM that is preallocated on a PM, and notify the load-balancer that the VM is ready to receive traffic.
- Stop running a VM that is running on a PM, and notify the load-balancer that the VM does not take traffic anymore. The VM stays allocated on the PM.

The system structure is illustrated in Figure 1. In this example, we see a system with 2-preallocation and four PMs. Each PM has one running VM, circled. Three of the VMs have an additional VM preallocated, and one is currently allocating a VM – loading it from network storage, marked with an arrow. Client queries are routed by the load balancer to the PMs running the relevant VMs.

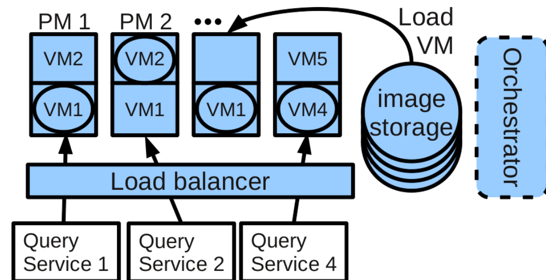

Figure 1: System architecture. Queries are directed by a load balancer to the relevant PMs. Each PM has up to 2 preallocated VMs, with one running (circled). VM images are loaded from an image storage. Preallocation and load balancing are managed by the orchestrator.

Note that when we allocate two VMs on each PM, we reduce the number of used PMs by half compared to classical allocation. However, it is possible that this preallocation is not be able to accommodate the incoming workload, should two services need the same PM concurrently.

3.2 Greedy Slack-Factor Preallocation

In this preliminary work, we use a simple greedy approach, allocating services in arbitrary order and with no explicit consideration of other services. We overview this preallocation strategy, and in the next section we show that even this simple heuristic leads to significant savings with our workloads.

Slack-factor preallocation works similarly to classical slack-factor allocation – each service is allocated a constant factor of α of its instantaneous load. However, here we count VMs that are collocated with others, possibly residing in PMs that currently run another VM.

The greedy slack-factor preallocation algorithm works as follows. In each step, the orchestrator checks for each service whether it has more VMs running than currently necessary, and stops redundant VMs, chosen uniformly at random. Next, for each service, it tries to run as many VMs as it currently needs, choosing greedily among idle PMs where the service is currently preallocated. Finally, for each service, it calculates the required number of machines to preallocate ($w_i^t \times \alpha$), marks redundant preallocations as such, and starts new preallocations as necessary. To preallocate, the orchestrator can either (1) use an idle machine with an empty slot, *i.e.*, with fewer preallocated VMs than the preallocation factor, (2) deallocate a redundant VM (as marked by its service), and allocate the new VM in its place, or (3) start a new PM and initiate preallocation on it. The pseudo-code is given in Algorithm 1.

Algorithm 1: Greedy preallocation algorithm

1 **repeat** (run/stop)
2   *done* $\leftarrow$ TRUE
3   **foreach** *service i* **do**
4         $missing \leftarrow w_i^t -$ currently running i-VMs
5   **if** $missing < 0$ **then**
6   Stop $(-missing)$ i-VMs
7   *done* $\leftarrow$ FALSE
8   **else if** $missing > 0$ **then**
9             $available \leftarrow$ idle PMs with i allocated
10   **if** $available > 0$ **then** *done* $\leftarrow$ FALSE
11   Start $\min(available, missing)$ i-VMs in allocated idle PMs
12 **until** *done* $=$ TRUE
13 **repeat** (allocate/deallocate)
14   *done* $\leftarrow$ TRUE
15   **foreach** *service i* **do**
16         $missing \leftarrow \alpha \times w_i^t -$ currently allocated i VMs
17   **if** $missing < 0$ **then**
18   tag redundant VMs
19   *done* $\leftarrow$ FALSE
20   **while** $missing > 0$ **do**
21   **if** *found idle machine with free slot* **then**
22   allocate VM i on it
23                 $missing \leftarrow missing - 1$
24   *done* $\leftarrow$ FALSE
25   **else if** *found idle machine with tagged VM* **then**
26   deallocate tagged VM from it
27   allocate VM i on it
28                 $missing \leftarrow missing - 1$
29   *done* $\leftarrow$ FALSE
30 **until** *done* $=$ TRUE
31 **foreach** *service i* **do**
32     $missing \leftarrow \alpha \times w_i^t -$ currently allocated i VMs
33   **if** $missing > 0$ **then**
34   start *missing* PMs
35   allocate VM i on new PMs

4 Evaluation

4.1 Workload

In this section we evaluate greedy slack-factor preallocation in flash-crowd workloads of multiple services.

Our model is based on studies of flash-crowds in small services and sites [3, 21, 8]. These analyses show that load regularly fluctuates with a *peak to average ratio* (*PAR*) of 1.5. In a flash-crowd event, load suddenly spikes, with a PARs of 10–200. The load increase from regular load to peak is roughly linear, taking a total of 10–30 minutes. After the load remains high for a while, it drops exponentially back to normal.

We construct our benchmarks based on this model. We use 50 services. Each is assigned regular workload, which is normally distributed with an average of 4 and a standard deviation of 1. When the service experiences a load spike, its load rises to a maximum of about 80 (PAR=20) in 30 minutes, maintains this load for another 30 minutes, and then exponentially drops back to the baseline. Load spikes occur with an average frequency of one per day per service. Figure 2 shows three out of the fifty load traces.

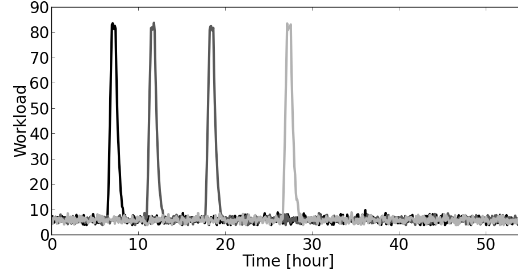

Figure 2: Workloads of three services with an average regular workload of 4 experiencing uncorrelated load spikes due to flash-crowd events with peak to average ratio of 20. Load increase is linear, over 30 minutes, maximum is maintained for 30 minutes, and then the load decreases exponentially.

4.2 Results

We now demonstrate the effectiveness of preallocation with the greedy slack-factor preallocation algorithm, using a homebrewed event-driven simulator. In each run, the orchestrator checks the per-machine load every 5 minutes, and makes necessary changes according to the greedy algorithm described above. Allocation takes 10 minutes [4, 20]. Deallocation and switching between machines is instantaneous, as we assume the time required to change routing and re-establish network connections is negligible.

We measure the number of PMs required to run multiple services in the cloud with different preallocation factors compared to classical allocation. These services must accommodate (almost) all incoming load. This is especially important for flash-crowd events, in which the service gets a sharp increase of attention, and must perform well. We therefore define a target overflow of 1%, and try to preallocate such that no service has an overflow larger than this target at any time.

For each preallocation factor, we run multiple simulation experiments with different slack factors, searching for the minimal slack factor for which the target is reached. We run simulations for 1, 2 and 3-preallocation, and count the number of PMs used with the optimal slack factor.

The results are depicted in Figure 3. Using a preallocation factor of 2 results in a savings of over 35% in both the average and the maximal number of running PMs. This means that with 2-preallocation, we need a data center 35% smaller, running an average of 35% less machines at any given time, compared to classical allocation. With 3-preallocation, the saving rises above 50%, i.e., requiring half the number of PMs.

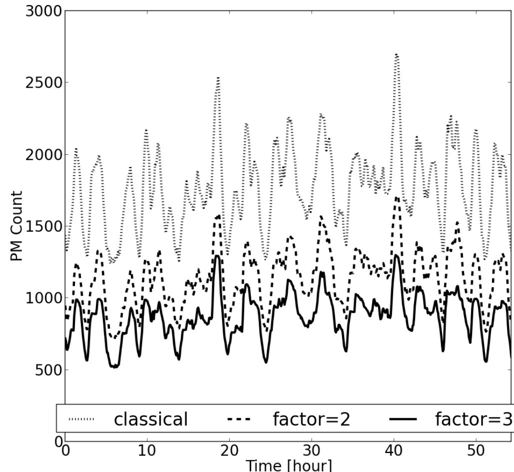

Figure 3: Number of PMs used as a function of time. The simulated data center hosts 50 services. The system provides per-service isolation, and up to 1% workload overflow at any time. Using preallocation factors of two and three results in a savings of over 35% and over 50% (respectively, both with respect to no preallocation) in the maximal number of PMs used, and in the average number of PMs used.

5 Practical Considerations

5.1 Slack-Factor Choice

In our simulation we measured the optimal slack-factor each client required by comparing the results achieved with different slack-factors for the same workload. While this methodology provides evidence that preallocation is effective, it is not practical, since the system has to choose the slack-factor in advance.

In practice, the slack-factor needs to be chosen based on an estimation of the incoming workload. This can be done using the same techniques currently used for performing allocation decisions (without preallocation), *e.g.*, based on analyses of historical workloads.

Note that the slack factor depends not only on the client requirements, but also on the requirements of other clients, and the preallocation factor of the system. The reason is that the machines on which a client preallocates VMs have to be available when its load spikes, and not be used by collocated VMs.

5.2 Billing

The question of billing for preallocated machine is an important one, since a preallocated VM consumes resources such as storage even if the VM is not running. One possible model is to charge for preallocating machines based on the probability that the VM becomes active. Preallocating more machines with a higher probability of becoming active, consequently, leads to a higher cost. This cost, however, is in principle smaller compared to fully allocating a PM to a VM. Another simple model is to charge only in the case the VM becomes active and charge for the period during which it is active.

Billing is an important topic for public clouds. For private clouds, it depends on the deployment model adopted. Companies offering cloud services often charge groups or organizations using the service. Such groups have their own budget and pay for cloud resources just as a client of a public cloud.

Independent of the type of cloud, given that techniques like preallocation save on the overall cost of ownership, it makes sense to charge less for preallocation compared to the a VM that fully utilizes a PM. In the case the VM image uses a small fraction of the PM storage, it might not even make sense to charge unless the VM becomes active.

6 Related Work

Kusic et al. [15] and Lin and Dinda [16] explore service allocation with multiple VMs running concurrently on the same PM. In [15], sleep modes are used, and the scheduler splits the resources of the running PMs among the VMs. In [16], the authors schedule a combination of interactive services and batch services. Apart from running multiple VMs concurrently on each PM, neither paper is concerned with the preallocation problem. Ghosh et al. [11] analyze power consumption in a cloud architecture with PMs in different sleep modes, each taking a different time to start a VM. They do not address the problem of how to perform preallocation.

Ganesh et al. [9] use frequency domain analysis to pack together services with workloads in different phases, so the overall peak of a rack is smaller than the sum of peaks. In our work we assume the workloads of the different services are independent and random.

The challenge of allocating workload without preallocation has been investigated thoroughly, see for example [10, 13]. As noted, sleep modes are complementary to our approach, as they reduce the number of running PMs, rather than their total number (for example [14], and see [6] for a comprehensive survey).

Note that the preallocation problem is not bin-packing, as the sizes of the objects (VMs) are identical and one has to decide how many copies of each VM should be allocated, and with which other VMs. We are unaware of a bin-packing variant that is analogous to the preallocation problem.

Optimizing preallocation is related to the problem of forming an expert team. There, we are given a team of experts, each with certain skills, and a problem that requires a certain set of skills. We have to find a set of experts that can solve the given problem, optimizing some parameter [5]. The setup of the problems is similar, as PMs are analogous to experts, the preallocated VMs to skills, and the workloads in each step to the problem to be solved by the experts. However, the expert team formation problem is that of choosing experts to solve a problem (allocating a workload to machines), whereas

the preallocation problem is that of allocating VMs to PMs (choosing the skills experts should learn).

Optimizing preallocation is also different from network routing and QoS optimization problems, where links have to be shared by different clients. In that case bandwidth can be continuously divided across clients, and flow can be instantaneously switched between links, requiring no preparation, as in the preallocation case.

7 Conclusion and Future Work

We introduced the concept of *preallocation* of virtual machines, allowing physical machines to serve as reserve for accommodating flash-crowds.

We have explored a simple greedy preallocation strategy and have shown that it potentially leads to important savings. Our results show a reduction of 35-50% of the physical machines compared to a data center in which each physical machine hosts one virtual machine (with 2- and 3-preallocation, respectively). More elaborate heuristics and machine learning techniques may increase the efficiency even further. To employ preallocation, PMs may need additional RAM and disk storage, however the vast reduction in PM count will surely lead to a significant reduction of the total cost of ownership.

The concept of preallocation can be vastly expanded. One direction to explore is utilizing multiple preallocation modes, where a VM image is loaded but not bootstrapped, bootstrapped and suspended to disk, or bootstrapped and suspended to memory as we do here. The different modes embody a tradeoff between the resources taken by the VM and the time needed to bring it to a running state.

Scaling stateful services, including storage tiers and replicated state machines is another challenge. Scaling such services requires careful planning to retain consistency, adding a cost to the action of scaling. This renders preallocation decisions even more complicated.

Combining preallocation with sleep modes adds yet another dimension to the problem. The challenge is to perform preallocation so that, when putting physical machines to sleep, the other machines have a good enough mixture of VMs to handle as much workload as possible. Sierra [22] is an example of a system that keeps all state available despite PMs being put in sleep mode.

Overall we have provided evidence that simple techniques used with resource allocation in cloud providers might lead to important benefits for both the providers and their clients by describing and evaluating preallocation. Our results, however, are not conclusive and some practical experience is necessary to demonstrate that the approach is really practical. In fact, the requirements of different provides might ask for variants of the preallocation strategy and as part of validation it is also important to contrast the results of different environments.

References

[1] virsh(1) — linux man page. http://linux.die.net/man/1/virsh, retrieved Mar. 4, 2012.

[2] xm(1) — linux man page. http://linux.die.net/man/1/xm, retrieved Mar. 4, 2012.

[3] ADLER, S. The slashdot effect, an analysis of three internet publications. http://www.astro.princeton.edu/~mjuric/universe/slashdotting/, retrieved Jan. 04.

[4] AMAZON. Amazon ec2 faqs – what can i do with amazon ec2? http://aws.amazon.com/ec2/faqs/#What_can_I_do_with_Amazon_EC2, retrieved Mar. 4, 2013.

[5] ANAGNOSTOPOULOS, A., BECCHETTI, L., CASTILLO, C., GIONIS, A., AND LEONARDI, S. Power in unity: forming teams in large-scale community systems. In *CIKM* (2010).

[6] BELOGLAZOV, A., BUYYA, R., LEE, Y., ZOMAYA, A., ET AL. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers 82* (2011), 47–111.

[7] DESHANE, T., SHEPHERD, Z., MATTHEWS, J., BEN-YEHUDA, M., SHAH, A., AND RAO, B. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA* (2008), 1–2.

[8] ELSON, J., AND HOWELL, J. Handling flash crowds from your garage. In *USENIX 2008 Annual Technical Conference* (2008).

[9] GANESH, L., LIU, J., NATH, S., AND ZHAO, F. Unleash stranded power in data centers with rackpacker. In *WEED* (2009).

[10] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI* (2011).

[11] GHOSH, R., NAIK, V. K., AND TRIVEDI, K. S. Power-performance trade-offs in iaas cloud: A scalable analytic approach. In *DSNW* (2011).

[12] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *MIDDLEWARE* (2006), Springer-Verlag New York, Inc.

[13] HARPER, R. E., TOMEK, L., BIRAN, O., AND HADAD, E. A virtual resource placement service. In *DCDV 2011* (2011).

[14] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. Napsac: design and implementation of a power-proportional web cluster. *ACM SIGCOMM Computer Communication Review 41*, 1 (2011), 102–108.

[15] KUSIC, D., KEPHART, J., HANSON, J., KANDASAMY, N., AND JIANG, G. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing 12*, 1 (2009), 1–15.

[16] LIN, B., AND DINDA, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *ACM/IEEE SC* (2005), p. 8.

[17] ONGARO, D., COX, A., AND RIXNER, S. Scheduling i/o in virtual machine monitors. In *VEE* (2008), ACM, pp. 1–10.

[18] OSVIK, D., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of AES. *Topics in Cryptology–CT-RSA* (2006).

[19] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS* (2009), ACM.

[20] STACK OVERFLOW. Is there a way to reduce time between azure deployment start and role onstart? http://stackoverflow.com/questions/10546624/is-there-a-way-to-reduce-time-between-azure-deployment-start-and-role-onstart, retrieved Mar. 4, 2013.

[21] TAK, B. C., URGAONKAR, B., AND SIVASUBRAMANIAM, A. To move or not to move: The economics of cloud computing. In *HotCloud'11* (2011), USENIX Association.

[22] THERESKA, E., DONNELLY, A., AND NARAYANAN, D. Sierra: practical power-proportionality for data center storage. In *EuroSys* (2011).