Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage

***Abstract*—In cloud computing, workloads that lease instances on demand get to execute exclusively for a set time. In contrast, workloads that lease spot instances execute until a competing workload outbids the current lease. Spot instances cost less than on-demand instances, but few workloads can use spot instances because of the variable leasing period. We present BOSS, a framework that uses spot instances to reduce costs for in-memory storage workloads. BOSS uses on-demand instances to create and update objects. It uses spot instances to handle read-only queries. BOSS leases instances from multiple sites and exploits varying prices between the sites. When spot instances stop abruptly at one site, BOSS places newly created objects at other sites, reducing the impact on response time. BOSS proposes a novel, online replication approach (1) avoids placing data at too many sites and (2) provides O(1.5)-competitive ratio under skewed cost distributions. Within a site, BOSS manages the tradeoff between savings and risks from replicating to spot instances. We implemented BOSS on top of Cassandra and deployed it on up to 78 instances across 8 sites in Amazon and Google clouds. With BOSS hosting TPC-W data, we spent $8 per hour on Amazon. We spent $55 per hour to use ElastiCache and $49 per hour to use only instances leased on demand. BOSS saved 85% and 84% respectively. Further, BOSS achieved 95th percentile response time within 10% of ElastiCache.**

I. Introduction

In-memory storage is vital for cloud services that respond to queries interactively, especially services with users spread all over the world. By 2018, the market for in-memory storage will exceed $13B, reflecting sustained growth of 43% annually [13]. Memcached, Cassandra and Redis are open source software packages commonly used to store data in memory [28]. Increasingly, clients prefer managed platforms that expose a networked API for creating, reading and writing data while handling replication and availability on behalf of their clients. Amazon ElastiCache [3], MemCachier [6], and MemCached Cloud [21] are examples of managed in-memory storage platforms.

Managed platforms can lease bundles of memory and CPU, called *instances*, from infrastructure-as-a-service (IAAS) clouds. For example, Memcachier leases from Amazon, Google, Rackspace and Azure [6]. The most common type of leases are for on-demand instances, where workloads get exclusive access to an instance for a set time. In-memory storage can lease on-demand instances for a long time to retain data stored in main memory. However, using on-demand instances is expensive for two reasons. First, on-demand prices include a surcharge for exclusive access because IAAS clouds can not use instances leased on demand to execute other (perhaps more profitable) workloads. Second, large in-memory storage capacity often leads to idle processing capacity. For example, in 2010, TripAdvisor spent 8% of its equipment budget on 350 GB of in-memory storage [12]. These servers could have processed 5B lookups per day but actually processed only 1B— 80% of the processing capacity was unused [12].

Spot instances provide another option for leasing resources from IAAS clouds. With spot instances, workloads get exclusive access to an instance only until a competing workload outbids the lease price. There is no surcharge for exclusive access because the IAAS cloud can revoke the lease at any time. As a result, Amazon EC2 spot instances can cost 90% less than on-demand instances [2]. However, to use spot instances, workloads must be allowed to stop abruptly at any time. Also, workloads shall be able to delay their execution when prices for spot instances rise. Prior research has addressed these challenges for data processing workloads that tolerate slow execution time [25]. These workloads, e.g., map-reduce jobs, include recovery techniques that can resume execution whenever they regain access to spot instances. If IAAS clouds provide a short warning, workloads can use virtual machine migration to pause and resume their work [24], [14]. This research could help in-memory storage avoid losing data when spot instances stop abruptly. However, other than storing data, spot instances are also used for query processing. It is challenging to process incoming queries when spot instances are unavailable without relying on pricey on-demand instances.

If in-memory storage platforms can not procure spot instances after failure, there are a few choices: (1) accept slower response time and possible data loss, (2) replace spot instances with on-demand instances, negating savings, or (3) move data and workload to other IAAS clouds where prices are cheaper. Spot prices on Amazon differ by up to 132% between its US East, US West, European and Asian sites [1]. (We use the word site to refer to an IAAS cloud). Further, spot market prices change over time at each site. Savvy workloads can reduce costs by constantly shifting data between sites to use the cheapest spot instance. However, too much data migration can increase costly inter-site network bandwidth.

This paper presents **BOSS**, a **B**lended **O**n-demand and **S**pot **S**torage framework for in-memory storage. In BOSS, leased on-demand instances process state-changing queries, e.g., object creation. Spot instances process read-only queries. This design reduces costs for in-memory storage serving mostly read-only queries and avoids data hazards (e.g., incomplete writes). The challenge for BOSS is the response time. When spot instances stop abruptly, re-dispatching read-only queries could overwhelm surviving instances. BOSS uses data replication between sites and within sites to anticipate the effects of variable leasing time. First, when BOSS creates objects, it places them at sites where spot instances are available. This shifts queries away from sites without spot instances. BOSS also balances query rates between sites by replicating heavily accessed objects to more sites than lightly accessed objects. This reduces the impact when spot instances at one site fail. However, placing objects at too many sites wastes in-memory storage on redundant data. We devised a novel, online replication approach that considers (1) the expected cost saving from replicating an object to more sites and (2) the spare capacity available for additional replication. Our approach profiles query rates for newly created objects and historical spot prices to model cost. If the statistical distribution of cost savings is exponential, our approach is $O(1+\frac{\omega}{|k_d|})$-competitive where ω is the reserved storage ratio and $|k_d|$ is the default replication factor. Our competitive ratio improves for any distribution more skewed than exponential.

Within a site, BOSS fully replicates data over spot and on-demand instances. BOSS exploits fast internal cloud networks that do not charge for intra-site bandwidth. When spot instances fail, their queries can be served by any other instances. BOSS allows users to trade response times for cost savings, like portfolio management. Users can use backup on-demand instances to mitigate risk. BOSS provides an efficiency frontier that helps users manage cost and risk.

We prototyped BOSS in Cassandra [17] and evaluated its costs on Amazon and Google clouds. In the prototype, $|k_d| = 2$. Thus, in the worse case, the performance of BOSS's inter-site replication is O(1.5)-competitive. On Amazon, our evaluation observes BOSS over 17 weeks under real spot instance prices. We compared BOSS to (1) ElastiCache, Amazon's managed in-memory store, (2) an auto-scaled in-memory storage implemented atop Amazon EC2 and (3) a recent research proposal based on virtual machine migration [14]. We deployed up to 78 Amazon EC2 instances across 8 sites. Results show that BOSS costs 85% less than ElastiCache while achieving comparable 95^{th} percentile response time (within 13%). BOSS cost 84% less than autoscaled in-memory storage using on-demand instances. Further, BOSS increases average utilization per leased instance by 60% compared to ElastiCache. Over 17 weeks on Amazon, BOSS never lost data. Google Cloud reduces savings by charging a fixed price for spot instances. We used BOSS on Google, costs were 56% lower than using only on-demand instances.

The contributions for this paper are:

- We present a novel framework for in-memory storage that blends reliable on-demand instances and cheap spot instances.
- We show that inter-site replication can mitigate the effects of spot instance failures.
- We prove that our online replication algorithm is at least $O(1+F(\frac{\omega}{|k_d|}))$-competitive.
- We present a design that achieves high throughput and handles spot instance failures by dispatching read queries to spot instances and state change queries to on-demand instances.
- We show that using an efficiency frontier of cost savings and response time variations can manage the risk/saving tradeoff.
- Using real spot instances from Amazon and Google, we show that our framework can significantly reduce costs compared to other managed in-memory storage platforms.

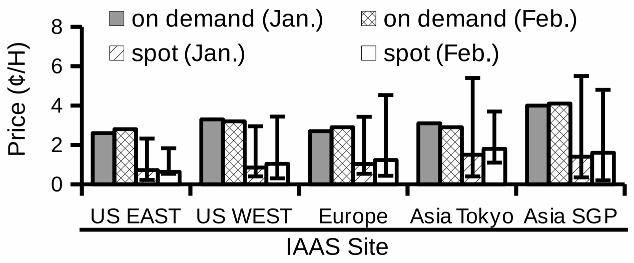


Fig. 1: Prices for instances leased on demand and spot markets in Amazon IAAS clouds.

II. Overview

BOSS provides Internet services with managed in-memory storage. It supports queries that create, read and write data objects, but controls the mapping of data objects to instances (i.e., it controls data replication). BOSS leases on-demand and spot instances from infrastructure-as-a-service (IAAS) clouds. Normally, these IAAS clouds are located at geographically diverse sites where regional factors, e.g., electricity prices, affect instance prices. BOSS assumes that each site supports:

1) High-speed internal networks that allow instances hosted in the same site to communicate without cost.
2) Instances can be leased on demand at any time.
3) Spot instances require winning bids. Bidders can lose many times before winning a lease. Further, spot instances can be stopped at any time.

Figure 1 plots the average lease price for on-demand and spot instances at 5 Amazon EC2 sites [1]. We show snapshots for two days (Jan. 30 and Feb. 30, 2015). On demand prices differed by 42% between US East and Asia Singapore, but within a single site, prices were stable. In contrast, US East provided the cheapest spot instances on Jan. 30 while Asia SGP had the cheapest spot instances on Feb. 30.

BOSS targets in-memory storage with these features:

1) Data objects are accessed frequently enough to exceed the CPU capacity of a single instance.
2) Objects are created and retired frequently. Total storage capacity needed for all objects is stable.
3) Read-only queries are more frequent than queries to create and update objects.

In-memory storage is usually reserved for data objects accessed very frequently. The most popular objects have peak query rates 12X larger than the average object [28]. BOSS assumes that objects with low access rates should be evicted to cheaper storage, e.g., SSD. Newly created objects are normally accessed more often than older objects, so LRU replacement policies can manage eviction and maintain stable storage capacity [28]. In addition to LRU, in-memory storage enforces

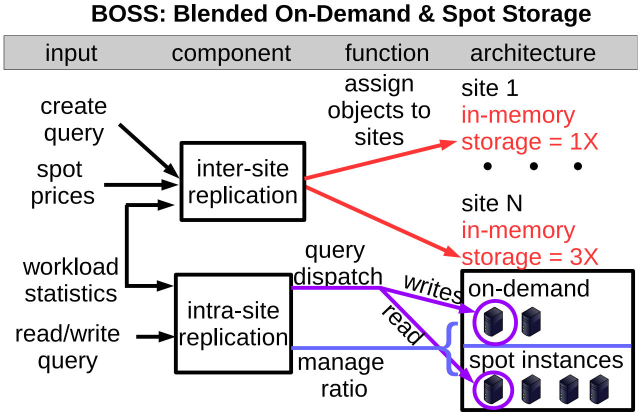


Fig. 2: The BOSS framework.

time-to-live (TTL) thresholds. In the Google cluster, more than 95% of objects are retired or evicted within 7 days [5].

Figure 2 highlights software that manages inter-site and intra-site replication in BOSS. Within a site, a query dispatch framework sends incoming queries that create or update objects to on-demand instances. This ensures high data availability. On-demand instances use chain replication [27] that provides scalable write throughput. Read-only queries are dispatched to spot instances. BOSS uses an efficiency frontier to help users manage the ratio of spot instances to on-demand instances. Spot instances use fast internal networks to fully replicate data stored on on-demand instances. Across sites, BOSS uses a cost model to find the best replication policy for each object. The cost model penalizes sites where spot instances are unavailable. Inter-site replication also uses an online algorithm to determine which objects deserve extra copies (for cost saving). The subsequent sections discuss inter-site and intra-site replication in detail.

III. Inter-site Data Replication

Figure 3 shows the life cycle of a data object in BOSS. When a query that creates an object arrives, BOSS applies Amazon's consistent hashing [9] to replicate the object to multiple sites. Each object must be copied to at least $|k_d|$ sites for availability. User can set $|k_d|$, or follow the default system configuration. BOSS profiles each object's query rate during this initial placement and fits a decay function. Note that, we use placement and replication interchangeably throughout this paper. Query rates are normally skewed, e.g., following exponential models. The profiled query rate and spot prices from the last 24 hours are fed into a cost model and online replication algorithm. This produces our final data placement. Objects stay at their final sites until their TTL times out.

BOSS avoids moving data between sites to reduce costly wide area bandwidth. After online replication, objects are placed at their final sites. BOSS has an increased cost when the final sites are chosen poorly. The only mechanism in BOSS to mitigate bad decisions is to recreate the object, e.g., after a TTL timeout. In Section V, we compare this design to a recent approach that moves objects when spot instances fail [14].

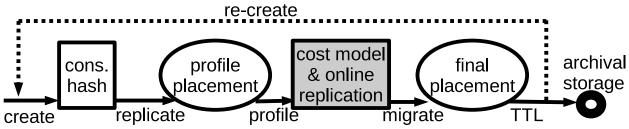


Fig. 3: Inter-site replication in BOSS. This paper focuses on the cost model and online replication algorithm

The rest of this section focuses on our (1) cost model and (2) replication algorithm.

A. Cost Model

Replication policies can increase storage costs by assigning objects to sites with highly priced instances. Also, policies that increase the frequency of inter-site communication increase storage costs. We model the total cost as a weighted function of (1) expense to store object at the replication sites and (2) latency between the sites. Equations 1–5 show the model:

$$P_i \triangleq P_{i,w} + P_{i,r} + G_i \tag{1}$$

$$P_{i,w} \triangleq \frac{\sum_t^T w_i^t}{T \times \bar{I}_{OD}} \times \frac{\sum_{j \in k} p_j(OD)}{|k|} \tag{2}$$

$$P_{i,r} \triangleq \sum_t^T \frac{r_i^t}{\bar{I}_{SP}} \min_{j \in k_i} p_j^t(SP) \tag{3}$$

$$L_i \triangleq (1-\tau) \times \frac{\sum_{t \in T} w_i^t}{T \times \bar{I}_{OD}} l_w + \tau l_r + l_{routing} \tag{4}$$

$$C_i(k) \triangleq \alpha P_i + \beta L_i \tag{5}$$

In these equations, i indexes the i^{th} data object and j indexes the j^{th} site. $C_i(k)$ is the total cost of serving object i under a replication policy k, where k is a set of host sites (e.g., {US East, US West, Europe, ...}). The size of the set $|k|$ is the *replication factor*. The total cost is a linear combination of financial cost (P_i) and inter-site latency (L_i) [20]. Parameters α and β are weight coefficients. The parameters $p_j^t(X)$ capture the price of instance type X at time t. The two instance types are SP and OD. Time t is the t^{th} interval in the TTL T. We use $p_j(X)$ to capture the average price over time T.

P_i captures the expense of leasing on-demand instances for writes $P_{i,w}$, spot instances for reads $P_{i,r}$, and the network bandwidth expense G_i of migrating data between sites. Let w_i^t be the query arrival rate for updates (writes) to object i. Also, $\bar{I}_{OD}$ captures the query processing rate per unit time t for on-demand instances. Thus, $P_{i,w}$ is the number of on-demand instances provisioned at each site to handle write queries. To be sure, BOSS fully replicates data across on-demand instances. G_i is the multiplication of allocated storage, the cost of network bandwidth and the frequency of migration. In BOSS, all objects are allocated the same size container (4KB). Each object is migrated at most $|k_d| + |k|$ times.

The cost of read-only queries is $P_{i,r}$. Unlike write queries, read-only queries are processed by only 1 spot instance across all sites. Further, BOSS can scale the capacity of spot instances. Thus, the cost to serve the object is the integral of the product of the number of spot instances needed for reads r_i^t/I_{SP} and cheapest spot instance price at time t. BOSS uses spot prices from 24 hours prior to estimate $p_j^t(X)$.

The lowest possible latency of one data operation L_i is the sum of the average write latency l_w, multiplied by the number of replicas (writes applied to all replicas), and the average read latency l_r, plus the routing delay $l_{routing}$. Note here, we calculate the routing delay $l_{routing}$ based on the average distance among all selected sites. Also, τ is the percentage of read-only queries.

B. Online Replication Algorithm

Each BOSS site receives a stream of newly created objects. After a profiling period, our online algorithm tries to reduce costs by carefully assigning objects to sites. The key insight is that: *For many objects, over-replication to to more than $|k_d|$ sites can reduce costs by serving read-only queries more cheaply, i.e., $\exists k \in \mathbf{k} : C(k_d) > C(k)$,* where $\mathbf{k}$ is the set of all replication policies. However, BOSS has limited capacity to over-replicate, i.e., $|k_{full}| \times \mathbf{I} > \Omega$. Here, k_{full} replicates data objects to all sites, $\mathbf{I}$ is total number of objects and Ω is the total in-memory storage capacity.

The challenge is to maximize cost savings within the capacity budget. Bin packing can solve this problem at a single site. However, with create queries streaming to all sites, bin packing algorithms would require BOSS to update spare capacity between sites every each time an object was updated—using costly inter-site communication. We preferred a distributed solution. BOSS allows each site to operate independently by over allocating space to over-replication. When a site chooses to over-replicate, it loses a replication slot at all sites. Each site is tasked with finding the objects with largest cost savings—even though some spare capacity is stranded. (Later, we show competitive guarantees despite internal fragmentation). Let k^o be the replication policy with minimum cost. Cost savings for object i are $C_i(k_d) - C_i(k^o)$. The online version of this problem tries to predict the largest objects from a stream [16]. Specifically, for each object, we decide to either (1) replicate to its lowest cost placement k^o or (2) stick to the default replication k_d. Note, we can get k^o by exhaustively computing Eq. 5 for all replication policies $k \in \mathbf{k}$.

BOSS uses the multiple choice secretary algorithm [16] (MCSA) to estimate the n_l objects with largest cost saving from n arrivals. First, BOSS prepares for n incoming objects by randomly selecting markers m_i from the binomial distribution $B(n, 1/2)$ recursively. In each recursion between markers, BOSS expects to select $\frac{n_l}{2}$ objects for over-replication. When n_l within a marker region is 1, BOSS observes $\frac{m_{i+1}-m_i}{e}$ objects, records the largest savings and then over-replicates the next object with greater savings. If the algorithm reaches the last marker with fewer than n_l objects, it chooses any remaining object with savings larger than the median of selected objects.

C. Performance Analysis: $O(1+\frac{\omega}{k_d})$-Competitiveness

BOSS uses MSCA to pick the top n_l data objects with largest cost savings, and over-replicates them across multiple sites. In this section, we prove that if the data access distribution follows an exponential distribution, the performance bound of the algorithm is $O(1+\frac{\omega}{k_d})$, depending on ω (i.e., the reserved storage factor) and $|k_d|$ (i.e., the default replication factor). First, we need to find the distribution of the cost.

Lemma III.1. *Assuming the data access distribution (denoted by x) follows an exponential distribution $x \sim Exp(\lambda)$, and the spot price history holds in the replication policy k, then the cost distribution of the replication policy $C(k)$ follows exponential distribution $C(k) \sim Exp(\mu)$, where μ is a linear function of λ.*

The spot price is hard to predict, BOSS can only know sites that have had very low spot prices. To estimate the cost of one replication policy, we assume the price history holds during this replication period. In this case, given a fixed replication policy, the cost of replication policy depends on the incoming reads and writes. We can estimate the read-to-write ratio from profiling. Thus, the total cost in Eq. 5 can be converted into a linear function of data accesses x. If the data access distribution x is an exponential distribution, then the cost distribution $C(k)$ is also an exponential distribution (If $x \sim Exp(\lambda)$, then $(ax) \sim Exp(\lambda/a)$). More importantly, if the cost distribution of any replication policy $C(k)$ is exponential, the achievable cost savings $C(k_d) - C(k^o)$ is also exponential. This claim can be proved by calculating the joint density function of two independent exponential distribution, which is also a density function of an exponential distribution. We define the competitiveness of BOSS, denoted by $\mathbf{f} : \mathbf{C}(BOSS) = (1+\mathbf{f})\mathbf{C}(OPT)$, where $\mathbf{C}(BOSS)$ and $\mathbf{C}(OPT)$ represent the total cost of using decisions from BOSS and the offline optimal solution, respectively. To find $\mathbf{f}$, there are two steps. In the first step, BOSS uses MSCA to pick the possible top n_l objects with largest cost saving without looking into the future objects. We need to find out the cost distance between the objects selected by MSCA and the actual top n_l objects.

Lemma III.2. *Given any set of non-negative real numbers ordered by value, and let us have $\mathbf{C}(\gamma)$ equals to the sum of the largest n_l elements in the set. The expected value of the elements selected by MSCA is at least $(1 - \ominus(\sqrt{1/n_l}))\mathbf{C}(\gamma)$.*

This is competitiveness of the MCSA online algorithm. Due to the limited space, we refer the proof in prior research [16]. Lemma III.2 provides the performance competitiveness between BOSS's decision and the decision (denoted by γ) of picking the actual top n_l objects. However, this decision may not be the optimal one. Due to misprediction, γ may not select the right set of n_l, which leads to wasted storage and thus less savings. Therefore, our second step is to find the competitiveness between γ and the offline optimal decision.

Theorem III.3. *If we only optimally replicate the top n_l objects, as the replication decision γ, the competitiveness of γ is $1+F(n_r)$, where $F(\cdot)$ is the CDF of normalized cost saving distribution and n_r is the ratio of objects that are* ***not*** *selected by γ.*

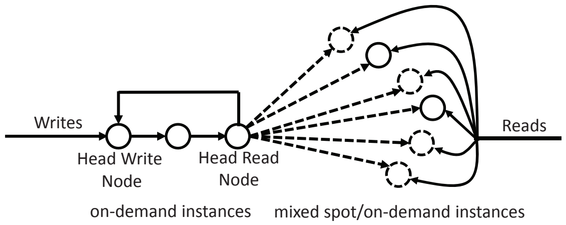


Fig. 4: The intra-site configuration design of BOSS. Dashed circle represents spot instances while solid circles are on-demand instances.

The cost savings missed by decision γ is from optimally replicating data objects in the set of $n_r\Omega$. In the worse case, γ may select a subset of objects but we may have enough spare space to store all data objects optimally. Thus, we can calculate the competitiveness by accounting how much more normalized savings that can be achieved if all data are replicated at their own k^o. In a normalized cost saving distribution, the rate parameter (denoted by v) of $F(\cdot)$ is in $(0,1]$, and the ratio of unselected objects $n_r = \omega/|k_d|$. At last, we have the competitiveness of BOSS in Theorem III.4.

Theorem III.4. *If the data access distribution is exponential, the competitiveness of BOSS is* $O(1+\dfrac{\omega}{k_d})$.

Proof. Using Lemma III.1, III.2, and Theorem III.3. We have

$$\begin{aligned}
&\mathbf{C}(BOSS) = (1+F(n_r))(1-\ominus(\sqrt{1/n_l}))\mathbf{C}(OPT)\\
&= (2-e^{-v\frac{\omega}{|k_d|}})(1-\ominus(\sqrt{1/n_l}))\mathbf{C}(OPT)\\
&= (2-e^{-v\frac{\omega}{|k_d|}}-\ominus(\sqrt{1/n_l})+e^{-v\frac{\omega}{|k_d|}}\times\ominus(\sqrt{1/n_l}))\mathbf{C}(OPT)\\
&(\text{Apply Taylor series expansion})\\
&\leq (2-1+v\frac{\omega}{|k_d|}-v^2\frac{\omega}{|k_d|}^2/2-\ominus(\sqrt{1/n_l}))\mathbf{C}(OPT),(v\leq 1)\\
&\leq (1+\frac{\omega}{|k_d|})\mathbf{C}(OPT)
\end{aligned}$$

□

In the worst case, if the spare capacity is running out and lots of data objects are created, then $w \rightarrow 1$. In our setup, we follow the replication configuration in Cassandra, in which $|k_d| = 2$. As a result, the theoretical competitive ratio of our prototyped BOSS is $O(1.5)$ in the worst case. This competitive ratio improves for any distribution that are more skewed than exponential. The skewness indicates that the ratio between cost savings from optimally replicating the top n_l objects and from optimally replicating all data objects is higher.

IV. Intra-site Replication

In Figure 4, BOSS lays out on-demand instances for high write throughput and spot instances for read throughput. Each instance runs a local in-memory storage that independently supports read, write and create operations. The BOSS intra-site replication protocol describes the prorogation of queries across instances. First, we describe the support for write queries. Clients route write queries to any on-demand instance. A hash function parameterized by the query payload determines a random order of the on-demand instances. BOSS then processes the query at each on-demand instance in order. The client can observe her write only after it is processed by the last on-demand instance [27]. Only if all on-demand instances within a site store the update, the client then observes a successful write, indicated by a per-object version number. The last on-demand instances in the write chain broadcasts the query to (1) spot instances within the site and (2) other sites hosting replicas. We use the gossip protocol to update (possibly failed) spot instances slowly over time [17]. At the beginning of each period, say 1 minute, all nodes randomly pair with neighbors and share the latest "gossip" on up-to-date data versions. This communication protocol helps the system to scale out and easy to configure for new instances.

Read-only queries are distributed across many spot instances. Clients can query any on-demand instance to learn the set of active spot and on-demand instances. Read-only queries should be routed to available spot instances first. However, if the response time for queries routed to spot instances exceeds 2X the service delay, clients can also route queries to on-demand instances. Clients are encouraged to randomly choose between available instances for reads.

BOSS provides causal consistency for clients that send write queries for an object to only 1 site and use version numbers for coordination. However, causal consistency degrades read throughput. Reads should be allowed to return stale data, following a LazyBase model [7].

A. Risk Management

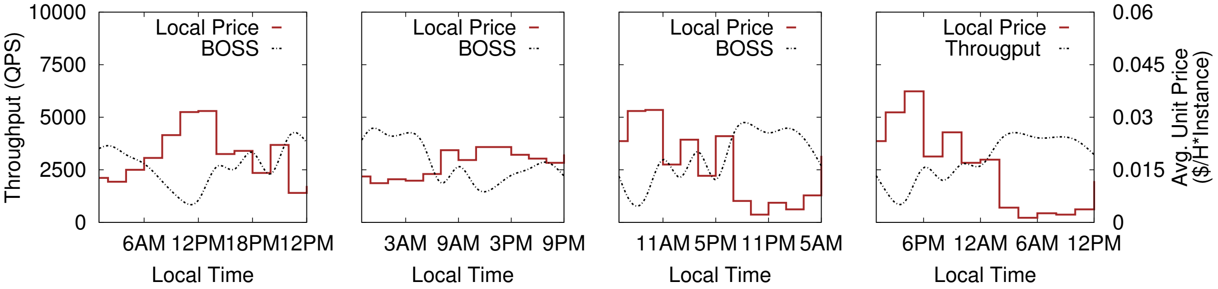
How many on-demand instances should BOSS lease? By design, it must lease enough on-demand instances to handle write throughput. However, when spot instances stop and the response time increases, clients can direct read-only queries to active spot and on-demand instances. Over 6-month, we observed only one on-demand instance stop abruptly. Additional on-demand instances mitigate the effect of losing spot instances. However, it also negates cost savings. To quantify this risk, we measure the performance delays from blended instances with different ratios of spot to on-demand instances. To be precise, we run offline experiments that capture average response time and cost savings during a short period for each spot-to-on-demand ratio. An mean-to-variation efficiency frontier plots response time standard deviation to cost savings mean. Any point on the frontier is an efficient operating configuration. Users can select the specific operating point that fits their desired risk. In Section V, we show an example of this frontier from multiple intra-site instance configurations. Our approach is analogous to portfolio management in the stock market. One can purchase low risk bond or high risk stock. Depending on one predefined risk (i.e., the response time variation in our case), there exists one instance mix that provides the highest expected return (i.e., cost savings).

B. Prototype

We prototyped BOSS as a Cassandra (v2.1.8) node image and deployed it to serve data processing workloads [17]. We

Name		HW Specification		Average Unit Price ($/H)							
	Type	vCPU	Memory	US East	US West	EU IRE	EU FFM	AP SGP	AP TYO	AP SYD	SA SP
m3.medium	on-demand	1	2GB	0.067	0.077	0.073	0.079	0.098	0.096	0.093	0.095
cache.m3.medium	Cache	1	2.78GB	0.09	0.094	0.095	0.103	0.12	0.12	0.12	0.115
m3.medium	Spot	1	2GB	0.0072	0.0085	0.0102	0.0119	0.014	0.0127	0.0103	0.011

TABLE I: On-demand, cache and spot instance profiles.



(a) US East (b) US West (c) EU IRE (d) AP SPG

Fig. 5: A 24-hour throughput of four sites under the local price variation. Data are recorded at the same time but x-axis is adjusted to the site's local time. QPS is query per second.

choose Cassandra to implement BOSS because of its rich APIs for supporting in-memory workloads and easy configuration for starting spot instances. Cassandra supports the gossip protocol to identify and explore connected instances within a single site. The default replication factor $|k_d|$ and the TTL in Cassandra are 2 and 7 days, respectively. The reserve ratio ω is reset at the beginning of each TTL to ensure we have enough space to store all data under the default replication. We have modified the *noodtool* to look up the private IP addresses in other sites so that BOSS can perform inter-site replication. The query dispatch is implemented in the Cassandra load balancer. To profile data objects and monitor the runtime performance, we use Amazon CloudWatch and the *noodtool status* command to capture instance utilization, network usage/delay, instance failure rate, and workload statistics. The price meter API from Amazon provides runtime unit prices, variations and total expenses of BOSS and other in-memory storage platforms. BOSS uses the bid price that would have provided 90% uptime in the previous month. However, a better bidding price model could help improve the performance of BOSS. We consider this as an extra layer on top of BOSS for future work, noting prior work [25], [24].

V. Evaluation

We evaluated BOSS on IAAS clouds offered by Amazon and Google that lease both spot and on-demand instances. Real clouds allow us to study BOSS as spot prices change in the real world. (To be sure, we can not control price changes in our evaluation.) Table I provides details about CPU, memory and prices for the instances used. For spot instances, we report the average winning bid is 0.0096$/H across all sites. For Amazon, we used 8 sites. Spot instances were 86–92% less than on-demand instances at the same site. The average lifetime of a spot instance was 36 minutes and prices between sites were not correlated. For Google, we used 8 global sites. We focus on Google pricing later in this section.

We compared BOSS to other systems to show that (1) BOSS costs less than widely used alternatives and (2) both inter-site and intra-site replication contribute to cost savings. Below, we describe baseline systems associated with each category above. For the remainder of this paper, we will refer to each baseline using the italicized words.

State of the Practice:

Default uses only on-demand instances for all queries. Consistent hashing is used to replicate data across sites. We enable autoscaling in Amazon EC2. When the instance is idle, Amazon checkpoints data to EBS and stops charging for the on-demand lease.

ElastiCache is Amazon's managed in-memory storage. It is used by Internet services like Netflix. We added consistent hashing to support replication across multiple sites. Amazon autoscaling also reduces costs for ElastiCache.

Migration mimics a recent approach that migrates virtual machines (VM) across sites when received termination warining [14]. This approach stops autoscaling by regularly sending state changes to a central server. Our version checkpoints internal Cassandra state (not the whole VM).

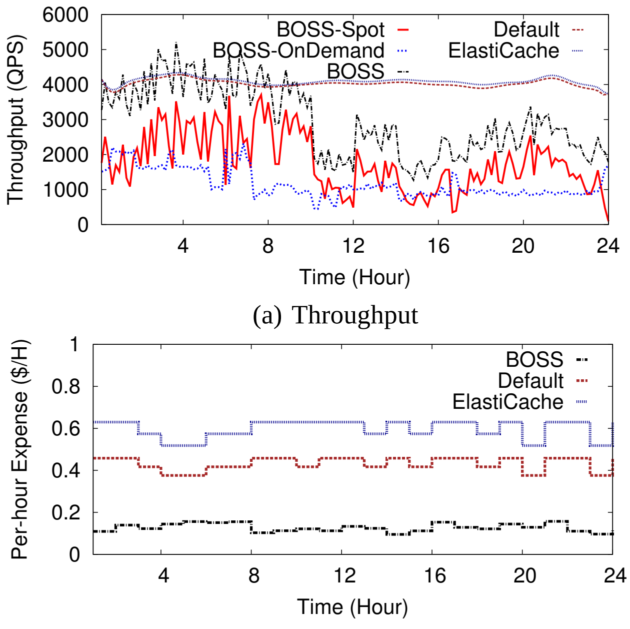
Alternative BOSS Designs:

InterOnly disables intra-site replication to spot instances. This approach shows cost savings for inter-site replication using only on-demand instances.

IntraOnly disables inter-site replication for low cost. It uses consistent hashing to place objects across sites. Like BOSS, IntraOnly uses on-demand instances to cover for failed spot instances.

Oracle discards data profiling and prediction in BOSS, using future knowledge about the trace to estimate the effect of perfect workload prediction.

We feed each site workloads based on Google's cluster cloud traces [5]. We implemented a latency-aware load balancer to route queries to the nearest site that hosts the data [22], [11]. It is important to note that nearest site load balance inflates the cost of serving read-only queries by using



(a) Throughput

(b) Average Expense Per Instance

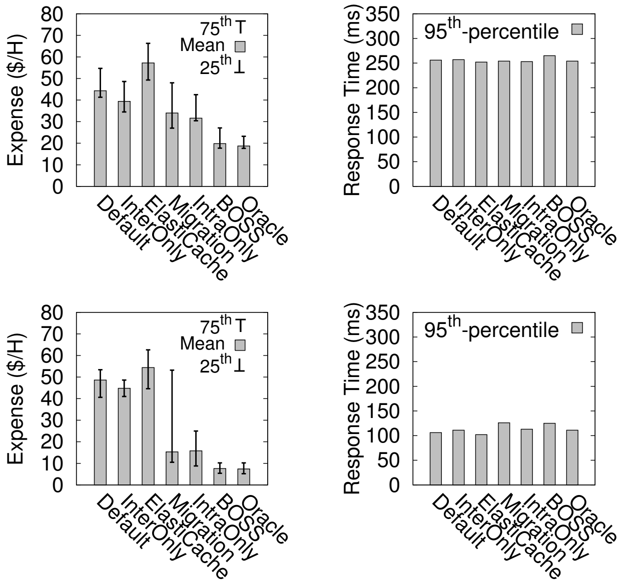
Fig. 6: Performance snapshots between BOSS and Amazon baselines in US West.

spot instances at more expensive sites. Our evaluation shows this cost does not negate the benefits of BOSS. The in-memory workloads used in our evaluation are:

WordCount is a MapReduce (MR) benchmark. It contains a wordcount MR program and a 10GB trunked Wikipedia data dump. This workload aims to test the performance for a large quantity of random key-value pair read.

Database contains TPC-W and simple update queries. It represents sequential reads on large database files. The update query writes randomly on popular data objects.

24-hour Performance Analysis. In Figure 5, we show throughput and average price per instance at 4 sites over 24 hours. The x-axis is adjusted to each site's local time zone. When it was daytime at US East (i.e., 8AM to 3PM), spot instance prices were high (e.g., the mixed price is 0.03 $/H). Data created during this time period was replicated to other sites instead of US East. As a result, the throughput at US East decreased from 4300 QPS (i.e., query per second) to 2000 QPS, while the throughput at EU Ireland and Asia Singapore increased 26% and 30%, respectively. We further illustrate the cost reduction impact of BOSS within a single site. Figure 6 illustrates the throughput and the expense of running BOSS in US West for 24 hours. To highlight the performance of BOSS, we compare it to *Default* and *ElastiCache*. From Figure 6(a), we observed that BOSS shifts the query rate significantly over time. Sometimes, BOSS can have more throughput than *Default* and *ElastiCache* (e.g., around hour 4), due to the performance boost from serving reads on more spot instances. However, if spot instances fail, reads are routed to the backup on-demand instances, causing the sharp throughput drop (e.g., around hour 8 to 10). By carefully managing the variable lease time of spot instances, BOSS can significantly reduce expenses. Illustrated in Figure 6(b), BOSS cuts the cost by 83% on average during this period. The average per-hour expense of BOSS is 0.17$/H. while *Default* and *ElastiCache* cost about 0.43 $/H and 0.6$/H, respectively. ElastiCache is more expensive than Default because the instance price is around 10% higher, as shown in Table I.



(a) Expense

(b) Response Time

Fig. 7: One month performance comparison using *database* (first row) and *wordcount* (second row).

Cost/Performance Comparison. Figure 7 shows one month performance and cost comparison between BOSS and all baselines. Shown in Figure 7(a), BOSS is 84% lower than Default, 66% lower than InterOnly, 85% lower than ElastiCache, and 50% lower than IntraOnly under the database workload. The cost savings increase under the read-only wordcount workload. In both cases, we observe that workload misprediction (an effect captured by *Oracle*) causes very small increase in cost.

On the performance side, we examine the 95^{th}-percentile response time of all queries as the latency metric. Figure 7(b) shows that BOSS has highest response time on both workloads, but the extra delay is within 13% and 10% of Default and ElastiCache, respectively. Although spot instances can drop frequently, BOSS's two-layer replication shields it from massive response time increases.

Figure 8 demonstrates the number of instance leased and the average utilization from these instances on different in-memory storage platforms. *Default* and *ElastiCache* are usually over-provisioned with more instances and thus much lower utilization. *InterOnly* helps reduce the number of on-demand instances. *IntraOnly* boosts per-instance utilization by offloading reads to spot instances but the total number of on-demand instances is still over-provisioned. BOSS consolidates on-demand and spot instances that lead to high utilization (above 85%). However, we observe that workload mispre-

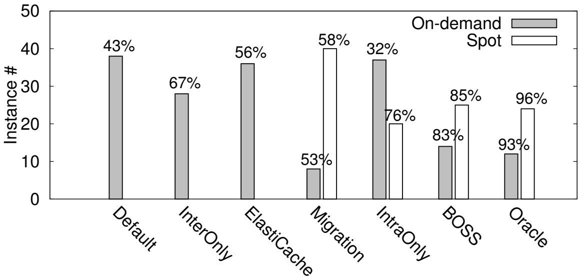


Fig. 8: The total number of instances leased using *database*. The number above each bar is the average instance utilization.

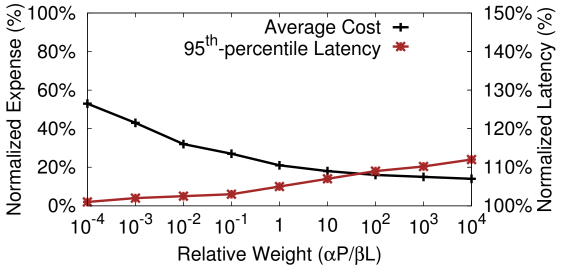


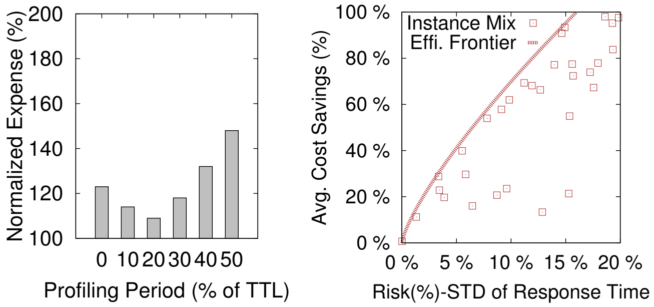
Fig. 9: The impact of tuning weight coefficients α and β using *database*. Note the x axis is in log scale.

diction can noticeably degrade the performance of BOSS, as comparing BOSS with *Oracle*. Better workload prediction algorithms could lead to increased resource utilization. It is worth noting that over 17 weeks of running BOSS on Amazon EC2, we have never experienced data lost.

Impact of weight coefficients. We have already shown that BOSS is good at reducing the cost of hosting data services using blended instances. There are two weight coefficients (i.e., α and β) in the cost model Eq.5 that affect the decision of picking sites and allocating instances. Instead of evaluating each one of them independently, we shown the performance of BOSS by tuning α and β to weight the two metrics (i.e., expense and latency), as shown in Figure 9. When we gradually increase α and decrease β, BOSS focus more on the cost reduction and less on the response time. As a result, the expense is monotonically decreasing when we look at the tail of the performance, the 95^{th}-percentile latency is increasing but slowly. The queuing delay due to the single vCPU is not affected at the tail performance.

Impact of Profiling Period and Risk Management. During the profiling phase, how long BOSS shall profile to achieve the reasonable accuracy while data is placed in sub-optimal sites is a concern. The longer time spent on profiling, the more we can predict the access pattern and workload statistics for certain data objects. We empirically set profiling time to 10%, 20%, 30%, 40% and 50% of TTL for the database workload. As demonstrated in Figure 10(a), the lowest cost was achieved at 30%. Normalized expense at each setting was 1.23, 1.15, 1.09, 1.19, 1.34 and 1.51, respectively. Long profiling periods were costly, reflecting the importance of inter-site replication.

Figure 10(b) plots the efficiency frontier for the database workload. We created the frontier offline with repeated experiments. Each point represents 20 experiments conducted with the same ratio of spot instances to on-demand nodes. The x-axis captures the variation of response time across the experiments. The y-axis captures mean cost savings. Points along the frontier maximize the mean-to-variation, thus are all good indicators for instance configuration.



(a) Profiling Period

(b) Risk

Fig. 10: Impact of (a) profiling period and (b) risk on BOSS. (a) is normalized to *Oracle*, and (b) is normalized to *Default*.

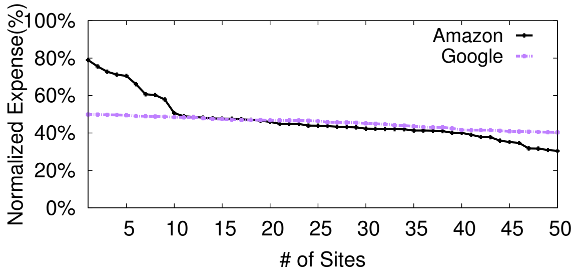


Fig. 11: Scale-out performance of deploying BOSS in Amazon and Google's cloud platforms. Normalized to *Default*.

Scale-out and Computational Overhead. To test the scale-out performance of BOSS, we use virtual site, which are duplicated sites based on the real site profiles from 8 distinct physical sites of Amazon EC2 and Google Cloud around the world. Each virtual site hosts 10 on-demand instances and up to 15 spot instances. The average prices for the on-demand instance are 0.085$/H and 0.05$/H, and for the spot instance are 0.0089$/H and 0.02$/H, from Amazon and Google, respectively. We plot the total expense of using BOSS to serve *database* workloads from 10 to 50 virtual sites. Compared to Amazon's spot instances, Google has a relative slower expense increase when scaling-out, as shown in Figure 11. All results show that BOSS can ensure high data availability by using on-demand instances while reduce costs by opportunistically using spot instances. As a framework, BOSS can be applied to increase the utilization of in-memory storage without understanding and defining workload types.

Another concern is the computing overhead of using the cost model Eq. 5 when scaling out. In our current prototype, we use exhaustive search to find the cost from the site combination with distinct site profiles. The computation time of searching the entire space of 8 physical sites is about 8 mins on a dual-core desktop computer. Note here, although we have 50 virtual sites in this experiment, as their profiles are from 8 distinct physical sites, their computation time is the same. The computation overhead from exhaustive search grows exponentially when the number of physical sites increases. However, hosting 8 physical global sites already exceeds the capacity of 95% cloud venders in the world [4]. In future work, we can use search heuristics [20] to keep the computation overhead linearized with the number of sites considered.

VI. Related Work

Today's databases and key-value stores commonly keep all data in main memory to support interactive data services [28]. Engineers start to build in-memory cloud infrastructure where a significant part, if not all, of the data fits in memory, such as Google spanner [8], HStore [15], RAMCloud [23], etc. BOSS contributes to this body of work with a focus on reducing costs by using blended instances.

When data is fully replicated, request dispatchers can balance the demands of popular objects. Tsitsiklis and Xu showed that these properties can be maintained with $O(lg(N))$ replication, provided replicas are created wisely [26]. The *content bottleneck* [18] occurs when the best replication policy can not be achieved because the spare storage capacity is limited. The inter-site design in BOSS applies partial replication across geo-diverse sites and exploits the cost variance from each location. The intra-site design is close to LazyBase, which serves reads and writes at degrees of staleness [7].

Load balancing between cloud sites reduces electricity costs, energy usage and carbon footprint [20]. Distributed online algorithms can improve such load balancing in practical scenarios when optimization is too challenging [19]. Under-used in-memory storage can also provide relief for overload application servers under adaptive load balancing [10]. AutoPlacer finds heavy hitters by solving complex optimization problems instead of using a distributed approach [22]. By exploiting the cost variance in spot markets, BOSS avoids replicating at at sites with high spot prices, and thus shifts workloads towards cheap sites without changing the load balancer.

VII. Conclusion

In this paper, we study the problem of lowering costs for in-memory storage. We design and implement BOSS, a two-layer framework that blends reliable on-demand instances and cheap spot instances. BOSS uses an online replication algorithm to optimally replicate data objects with largest possible cost deduction. At the intra-site, BOSS uses on-demand instances to ensure high data availability while opportunistically using spot instances to reduce costs. We prototyped BOSS in Amazon and Google IaaS platforms. Results show that using blended instances can significantly reduce the cloud expense (84%) without noticeable performance degradation (less than 13%).

References

[1] Amazon ec2 pricing. http://aws.amazon.com/ec2/pricing/(Last visited in July 2015).

[2] Amazon ec2 spot instances. http://aws.amazon.com/ec2/purchasing-options/spot-instances/ (Last visited in July 2015).

[3] Amazon elasticache. http://aws.amazon.com/elasticache/(Last visited in July 2015).

[4] Data Center Efficiency Assessment. https://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf.

[5] Google Cluster Service Traces. code.google.com/p/googleclusterdata.

[6] MemCachier Status. http://status.memcachier.com/.

[7] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. Lazybase: Trading freshness for performance in a scalable database. In *ACM EuroSys*.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX OSDI*, 2012.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[10] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch. Softscale: Stealing opportunistically for transient scaling. In *Proceedings of the 13th International Middleware Conference*.

[11] A. Ganesh, S. Lilienthal, D. Manjunath, A. Proutiere, and F. Simatos. Load balancing via random local search in closed and open systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):287–298, June 2010.

[12] A. Gelfond. Tripadvisor architecture - 40m visitors, 200m dynamic page views, 30tb data. http://highscalability.com, June 2011.

[13] I. Grand View Research. In-memory computing (imc) market analysis, market size, application analysis, regional outlook, competitive strategies and forecasts, 2014.

[14] X. He, P. Shenoy, R. Sitaraman, and D. Irwin. Cutting the cost of hosting online services using cloud spot markets. In *ACM High-Performance Parallel and Distributed Computing*.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2), 2008.

[16] R. Kleinberg. A multiple-choice secretary algorithm with applications to online auctions. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 630–631. Society for Industrial and Applied Mathematics, 2005.

[17] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[18] M. Leconte, M. Lelarge, and L. Massoulie. Bipartite graph structures for efficient balancing of heterogeneous loads. In *Proc. of SIGMETRICS*, 2012.

[19] M. Lin, Z. Liu, A. Wierman, and L. L. H. Andrew. Online algorithms for geographical load balancing. In *Proceedings of the 2012 International Green Computing Conference (IGCC)*, IGCC '12, pages 1–10. IEEE Computer Society, 2012.

[20] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew. Greening geographical load balancing. *ACM SIGMETRICS*.

[21] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook, 2013.

[22] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *USENIX ICAC*, 2013.

[23] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *USENIX FAST*, 2014.

[24] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, page 16. ACM, 2015.

[25] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *IEEE INFOCOM*, 2012.

[26] J. N. Tsitsiklis and K. Xu. Queueing system topologies with limited flexibility. *SIGMETRICS Perform. Eval. Rev.*, 41(1), 2013.

[27] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, 2004.

[28] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 27, July 2015.