Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud

Cheng Wang *†, Bhuvan Urgaonkar, Aayush Gupta‡, George Kesidis and Qianlin Liang
†VMware Inc., Penn State University, ‡IBM Research Almaden
†wangcheng@vmware.com, bhuvan@cse.psu.edu, ‡guptaaa@us.ibm.com, {gik2, qxl5068}@psu.edu

Abstract

In order to keep the costs of operating in-memory storage on the public cloud low, we devise novel ideas and enabling modeling and optimization techniques for combining conventional Amazon EC2 instances with the cheaper spot and burstable instances. Whereas a naturally appealing way of using failure-prone spot instances is to selectively store unpopular ("cold") content, we show that a form of "hot-cold mixing" across regular and spot instances might be more cost-effective. To overcome performance degradation resulting from spot instance revocations, we employ a highly available passive backup using the recently emergent burstable instances. We show how the idiosyncratic resource allocations of burstable instances make them ideal candidates for such a backup. We implement all our ideas in an EC2-based memcached prototype. Using simulations and live experiments on our prototype, we show that (i) our hot-cold mixing, informed by our modeling of spot prices, helps improve cost savings by 50-80% compared to only using regular instances, and (ii) our burstable-based backup helps reduce performance degradation during spot revocation, e.g., the 95% latency during failure recovery improves by 25% compared to a backup based on regular instances.

CCS Concepts **• Computer systems organization → Cloud computing**

Keywords spot instance, burstable instance, in-memory caches, public cloud

* Part of this research work was done when the first author was a Ph.D. student at Penn State University.

EuroSys'17 April 23-26, 2017, Belgrade, Serbia
© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064220

1. Introduction and Motivation

In-memory storage is crucial to many public cloud-based interactive applications ("tenants") for meeting their stringent latency and throughput requirements. Many such tenants exhibit temporal variations in workload features such as (i) request arrival rates, (ii) content popularity, and (iii) working set size. Consequently, *dynamic resource allocation* is crucial for operating them cost-effectively on the public cloud where costs are based on resource usage.

Not surprisingly, adapting traditional dynamic resource allocation strategies - studied in the context of in-house, private computing environments - to the idiosyncrasies of the public cloud is an active area of research (see Section 6). We find providers employing myriad innovations in pricing and resource guarantees (i.e., service-level agreements) to better monetize their otherwise under-utilized IT infrastructure. The result has been the emergence of "cheap" virtual machine (instance) types whose lower prices are accompanied by reduced "effective capacity" (i.e., capacity actually offered to the tenant as opposed to the advertized maximum).

These cheaper instances come in two main classes. The first class consists of revocable instances that may be selectively reclaimed by the provider in favor of the more expensive conventional (regular) instances when capacity needs of the latter go up. Amazon EC2's spot instances (2009), are the most well-known examples of revocable instances. More recently (ca. 2013), Google Compute Engine (GCE) has also offered pre-emptible instances which, despite operational differences from EC2 spot instances, similarly offer lower prices for poorer availability. The second class of cheaper instances are much more aggressively multiplexed than regular instances on overbooked servers. They come in smaller sizes and are known to exhibit significant capacity dynamism at as fine a time-scale as a few seconds or minutes.

Although spot instances have been extensively explored for cost-effective resource procurement, most of the studies only focus on batch processing jobs and use proactive checkpointing and/or reactive live migration to deal with spot revocations [13, 19, 39, 51]. However, when applied to in-memory storage, these may fail to offer satisfactory performance. To address this, some recent efforts [34, 49, 50] consider active replication: the cache contents are replicated across geo-distributed / weakly correlated spot markets for fault-tolerance, and all the replicas can be used to serve the requests for better performance. We would like to explore a complementary but fundamentally different design point based on the following ideas: (i) given that highly skewed content popularity distribution is the norm, we mix "hot" (popular) and "cold" (unpopular) content across spot and on-demand instances in order to strike a balance between performance and high resource utilization (which translates to lower costs), (ii) to reduce the occurrences of spot bid failures, we build scalable novel data-driven spot prediction models, which are then used for the dynamic procurement of spot instances, and (iii) to maintain performance goals in the event of a spot revocation, we design a passive backup based on newly emergent burstable instances whose ability to offer much higher CPU and network bandwidth per unit main memory compared to regular instances makes them ideal candidates for this purpose. To our knowledge, (iii) represents a novel use case for burstable instances. We use memcached as our target in-memory storage application for system design and implementation.

Contributions:

We make the following contributions.

- *Data Analysis, Optimization, and Algorithm Design*: We offer novel ways of combining regular on-demand, spot, and burstable instances advised both by the needs/properties of memcached, as well as the relative strengths and weaknesses of these instance types.
- *System Design and Implementation*: We offer a complete implementation of a dynamic resource allocation mechanism for memcached and a working prototype on EC2. The core components of our system include a global controller that periodically collects/predicts workload properties and conducts online optimization for resource allocation, a load balancer that dispatches requests to the appropriate nodes based on popularity and carries out failure recovery when spot revocation occurs, and a key partitioner that classifies the keys into hot and cold keys for our hot-cold mixing.
- *Evaluation*: We present an empirical evaluation of our ideas combining real-world and synthetic workloads as well as simulations and live experiments on our EC2 prototype. Our key results and lessons are: (i) our spot modeling outperforms the commonly used baselines by offering fewer spot revocations, (ii) our hot-cold mixing, informed by our modeling of spot prices, helps improve cost savings by 50-80% compared to only using regular instances, and (iii) our burstable-based backup helps reduce performance degradation during spot failures, e.g., the 95% latency during failure recovery improves by 25% compared to a backup based on regular instances.

2. Background

2.1 Memcached

memcached is a widely used distributed in-memory key-value cache [27]. In addition to being used by many popular web applications (Facebook, LiveJournal, Wikipedia, Twitter, Flickr, to name a few), it is also very popular among smaller, cost-conscious tenants that form our focus. Instead of running its own memcached on infrastructure-as-a-service (IaaS) instances (like we consider), a tenant may instead use a "managed" (i.e., software-as-a-service or SaaS) in-memory storage service built using memcached. One finds examples of such SaaS offerings both from IaaS providers themselves (e.g., Amazon EC2's ElastiCache, Microsoft Azure's RedisCache) and from other entities that procure instances from these IaaS providers (e.g., Redislabs' Memcached cloud [28], Heroku's MemCachier [29]). Our techniques could be useful to the latter type of tenants as well.

In a typical configuration, memcached employs a cluster of nodes (instances or physical machines) for housing its cache contents with a load balancer orchestrating data placement, client request forwarding, failure handling, and data replication (if used). In high-throughput environments, memcached clusters are routinely composed of 100s to 1000s of nodes, and the load balancer may itself be replicated for scalability and fault tolerance. memcached uses consistent hashing to map both keys and nodes to points on the same hashing ring. Each instance is responsible for hosting keys whose hash values fall into the interval mapped to it. Consistent hashing has been chosen for its well-known benefits: it maps data objects to the same cache node as far as possible, which is particularly useful when adding/removing nodes. These properties allow a memcached cluster to be dynamically scaled up or down. Given the time-varying nature of many memcached workloads, these properties make it well-suited for dynamically adjusting (auto-scaling) its instance usage to workload conditions on a public cloud. memcached implements options whereby its cache contents may be replicated for redundancy to improve both service availability and performance. Finally, it employs a least-recently used (LRU) cache eviction policy. Read requests for evicted or non-existent keys are served by the back-end.

Because spot instances may be revoked, not all workloads can benefit from their lower prices. memcached may be a good candidate for using spot instances. The failure of a memcached node does not affect application correctness if all data is persisted in the back-end. Similarly, a failure does not make the overall service unavailable but may slow it down (depending on how many requests now need to be serviced from the slower back-end). It is crucial that the cost reduction by using spot instances not result in a significant loss of performance. Of course, acceptable levels of performance loss would be tenant-specific. We assume that the tenant has mechanisms for carefully considering

Instance type		Per unit resource price		Smallest size		CPU or net. bw to RAM ratio	
		vCPU ($/vCPU*hour)	RAM ($/GB*hour)	vCPU	RAM (GB)	vCPU/GB	Mbps/GB
Regular	On-demand (OD)	0.0397	0.0057	1	3.75	0.13-0.53	18-146
	Reserved	26-37% cheaper than OD (equal size)		1	3.75	0.13-0.53	18-146
Spot		70-90% cheaper than OD (equal size)		1	3.75	0.13-0.53	18-146
Burstable	Base capacity	0 (see caption)	0.013	0.05	0.5	0.075-0.1	70
	Peak capacity			1	0.5	0.25-2	125-1000

Table 1: Comparisons of important aspects of regular, spot, and burstable offerings from Amazon EC2. The unit prices are calculated by using linear regression models ($R^2 = 0.99$). Surprisingly, network bandwidth does not play a role in our regression models; for burstables, neither CPU nor network bandwidth figure in the pricing model and the instance price is perfectly proportional to the RAM capacity.

and specifying this. One way of doing this could be based on comparing against reasonable baselines that operate the application on a well-provisioned cluster comprising highly available nodes; see Section 2.3.

Finally, our design targets read-heavy workloads. Evidence from real systems suggests this is an important class worthy of the techniques we develop for cost-efficacy. For example, consider the Facebook workload USR [1] where 99.8% of requests are read operations. We leave it to future work to extend our system to improve write performance, possibly by adapting techniques from prior work, e.g., using a small amount of on-demand instances (highly available) to serve write requests [50] faster than in our system where we write-through to a persistent back-end.

2.2 EC2 Instance Offerings

We find it useful to consider the following first-order classification of EC2 instance types: (i) regular, by which we mean conventional on-demand (i.e., not including burstable) and reserved instances that offer high availability and near-fixed resource capacities, (ii) spot, and, (iii) burstable instances (with guaranteed base capacity plus variable capacity determined based on the tenant's available resource tokens). Within each of these classes, there is further variety. We do consider multiple resource capacity sizes (EC2 offers about 40 different sizes for all of on-demand, reserved, and spot) in Section 4 but ignore other second-order price vs. capacity/performance trade-offs (e.g., "compute-optimized" vs. "memory-optimized" vs. "IO-optimized").

Price per unit resource: We find that a linear regression model is able to offer an almost perfect explanation of EC2 on-demand instance prices as functions of the number of vCPUs and RAM capacity. As an example, for 25 commonly-used on-demand instance types from the region of US West (price data obtained in October, 2016), the model $p = 0.0397 \cdot c + 0.0057 \cdot m$ explains instance prices with R-squared equal to 0.99, where p, c, m are the hourly instance price, number of vCPUs and amount of RAM (GB), respectively. Other classes may have slightly different coefficients but are also explained equally well. A similar model for burstable instances expresses their prices perfectly solely in terms of their RAM capacities. The per unit resource prices reveal interesting relative strengths for cost-efficacy and are used in our optimization in Section 4. Although the instance prices do change across regions and over time (although at a coarse time granularity, e.g., month, year), and the exact trade-offs may change, our techniques should apply equally well.

CPU and network bandwidth per unit RAM: For each instance type, we denote as "vCPU/RAM" and "Network BW/RAM" the CPU capacity in units of a regular EC2 vCPU (could be fractional as for burstables) and the network bandwidth (Mbps) associated with per unit RAM capacity (GB), respectively. High values for these ratios are desirable features for instances used in our passive backup (to be described in Section 3.3). The key observation we make is that, when operating at their peak capacity (the timing of which may be carefully controlled by the tenant itself), burstable instances offer much higher ratios for every dollar invested than regular or spot instances.

Table 1 summarizes the above trade-offs.

2.3 Baselines and Shortcomings

We now present several "baselines" for procuring instances for a memcached workload. These baselines represent a combination of (a) salient ideas from literature on dynamic resource allocation as well as (b) recent related work on memcached-specific resource procurement on the public cloud (see Section 6 for details). We use the baselines as strawman approaches in the following ways: (i) to identify and illustrate shortcomings in the state-of-the-art, (ii) to motivate important aspects of our system design, and (iii) as points of comparison in our experimental evaluation (Section 5) to quantify the benefits offered by our proposed approach. The first 3 rows of Table 4 summarize our baselines.

On-demand instances only: A natural first baseline is based on only using the highly available regular instances. Although reserved instances are 26-37% cheaper than their on-demand counterparts, their efficacy for unpredictable workloads is questionable since they require an upfront commitment for usage lasting 1-3 years. That is, in the absence of significant long-term predictability in resource needs, using reserved instances may be a "high-risk" proposition [46]. Therefore, we restrict our baseline to on-demand instances (billed at an hour's timescale). Using only on-demand instances would eliminate the complexity of bidding for spot instances and reacting to their revocations and the corresponding performance degradation. Within this, we consider two baselines. The first of these called ODPeak provisions for the peak resource needs at all times, whereas the second approach ODOnly modulates its allocation of on-demand instances to match the dynamic needs. Both baselines are computed offline (or equivalently represent hypothetical scenarios with perfect knowledge of the workload). ODOnly represents a state-of-the-art dynamic resource allocation (autoscaling) technique being adapted to our problem.

Popularity-aware data separation among on-demand and spot: memcached workloads (and storage system workloads in general) usually exhibit skewed data popularity distributions wherein a small fraction of the overall content ("hot") receives most of the requests with the remaining being relatively unpopular ("cold"). Much work exists on accurately and efficiently identify hot vs. cold data (including extensions for more fine-grained classification) in an online manner [52] and we assume such a technique is available to our system. This motivates our second class of baselines wherein we wish to combine spot instances with on-demand instances as follows: use cheaper spot instances for storing cold content and the more expensive on-demand instances for storing hot content. Clearly, an approach using such "hot-cold separation" to combine spot and on-demand instances is likely to offer significant cost savings compared to ODOnly. Note that, although based on a straightforward idea, this hot-cold separation is novel to our knowledge. More importantly, it is complementary to other recent proposals for operating in-memory caches cost-effectively on the public cloud, e.g., [50] - see a detailed discussion in Section 6. We refer to this approach as OD+Spot_Sep in our evaluation.

Although appealing due to its potential cost savings over using only on-demand instances, this class of baselines presents us with three important challenges.

- *Resource wastage*: Since hot data only takes a small portion of the overall working set but receives a large fraction of the requests, on-demand instances that hold the hot data would need more CPU/network resources but much less memory capacity than spot instances that hold the cold data, which may lead to resource wastage due to the rigid instance offerings of current public clouds. For example, as we show in detail in Section 5, using hot-cold separation for a scaled wikipedia workload employs 6 on-demand and 4 spot instances during the hour with the peak intensity. The average CPU utilization of the spot instances is a poor 18% whereas the memory occupancy of the on-demand instances is only 25%. The scheme on bottom-left of Figure 3 offers an illustration of such resource wastage.
- *Spot instance selection*: Most prior studies [16, 24, 35, 36, 39, 40] use empirical cumulative distribution function (CDF) of historical spot prices for spot price prediction. Such CDF-based models end up discarding valuable temporal information about the continuity of the spot price staying below different bid values, which may lead to poor decision-making. For example, as we show in Figure 8, although there are frequent bid failures under bid$_1$, the CDF-based approach still fails to anticipate bid failures since the CDF of spot prices does not vary much over time. We refer to this baseline as OD+Spot_CDF.
- *Spot revocations*: A final challenge is, of course, to be able to recover from spot instance revocations effectively. Prior studies mainly focus on state migration upon receiving a spot revocation from EC2 (either purely reactive or based on proactive checkpointing) [13, 19, 39, 51]. However, this may not be suitable for in-memory caches due to the resulting degradation in response times during the recovery process.

3. Key Concerns and Ideas

Assuming extensive prior work on predicting resource needs and identifying hot vs. cold content can be leveraged, our solution must then address the following three issues:

- *Procurement*: how many spot and on-demand instances (and what sizes) to procure? for spot instances, what bids to place and on which markets?
- *Usage*: what portion of the overall workload to place on procured spot and on-demand instances?
- *Recovery*: how to ensure that we recover quickly from a spot revocation? how to ensure that any adverse impact on performance due to such failure and the overheads of recovery is kept within acceptable limits?

3.1 Procurement: Exploiting Short-Term Temporal Locality

In order to answer the first set of questions, a tenant needs to model well the following two properties both conditioned on specific bids values (chosen from a small set of pre-selected values): "lifetime of a spot instance" and "average spot price during lifetime." Although an empirical distribution of historical spot prices would be able to predict these properties, it would fail to capture service contiguity (recall discussions on the shortcomings of the CDF-based baseline in Section 2.3).

We devise scalable data-driven models for these properties. We model as a random variable $L^s(b)$ the length of a *contiguous* period during which the spot price in market s is less than or equal to a bid b, which is an estimation of the upper bound of

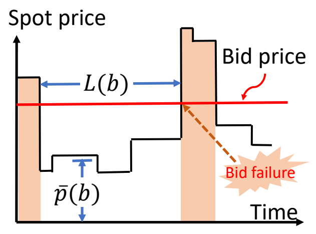

Figure 1: The lifetime of a spot instance and average spot price during lifetime.

	Bid	$f^s(b)$	$\xi^s(b)$	$f^s(b)*$	$\xi^s(b)*$
m4.large-c	$0.5d$	0.12	0.08	0.29	0.14
	$1d$	0.07	0.07	0.09	0.11
	$2d$	0.02	0.09	0.02	0.11
	$5d$	0.02	0.09	0.02	0.11
	$10d$	0.02	0.09	0.02	0.11
m4.large-d	$0.5d$	0.10	0.06	0.62	0.23
	$1d$	0.10	0.07	0.50	0.22
	$2d$	0.10	0.10	0.45	0.22
	$5d$	0.13	0.12	0.42	0.23
	$10d$	0.07	0.19	0.34	0.25

Table 2: The assessment metrics $f^s(b)$ and $\xi^s(b)$ under different (market,bid) pairs with history window of 7 days. "-c" and "-d" represent the markets of us-east-1c and us-east-1d. $f^s(b)*$ and $\xi^s(b)*$ are computed based on predictions of $L^s(b)$ and $\bar{p}^s(b)$ using CDF of spot prices.

the lifetime of an instance procured using bid b[1]. We denote as $\bar{p}^s(b) = E[p^s_t|L^s(b)]$ a random variable for the average spot price p^s_t in market s during a period when the bid b is successful, which serves to estimate the cost of a spot instance procured by placing a bid b. Figure 1 illustrates our definitions of $L(b)$ and $\bar{p}(b)$. Our prediction assumes temporal locality[2] over a recent sliding time window of multiple time-slots for making predictions of $L^s(b)$ and $\bar{p}^s(b)$. Large $L^s(b)$ and small $\bar{p}^s(b)$ imply long service continuity and low costs, thereby encouraging the use of spot instances using bid b. We use a small percentile (e.g., 5th) of the recently constructed distribution of $L^s(b)$–denoted as $\hat{L}^s(b)$–as our prediction in the ongoing horizon. The reasoning behind this choice is that if the statistical properties of $L^s(b)$ do not change much over the sliding window, we expect that with a very high probability, bid b would be successful for at least $\hat{L}^s(b)$ time units. We use the average of $\bar{p}^s(b)$ during the sliding window as its predictor (denoted as $\hat{\bar{p}}(b)$).

Validation: To evaluate our predictors, we introduce the following assessment metrics. We say that an *over-estimation* of $L^s(b)$ has occurred when $\hat{L}^s(b) > L(b)$. This represents a scenario wherein the tenant was likely overly ambitious in using spot instances. We further define $L^s(b)$ *over-estimation rate* as the fraction of $L^s(b)$ predictions that result in over-estimation, denoted as $f^s(b)$. The assessment metric for $\hat{\bar{p}}^s(b)$ should capture the extent of its deviation from actual values. We compute $\xi^s(b) = (\bar{p}^s(b) - \hat{\bar{p}}^s(b))/\bar{p}^s(b)$ and define as *relative deviation* of $\bar{p}^s(b)$ the mean value of $\xi^s(b)$ for all occurrences of $\bar{p}^s(b)$ in the sliding window. Lower values are better for both metrics.

[1] Clearly, the lifetime defined in this manner could depend intimately on the time when a bid is placed. We do not consider this complexity in our work because it is conceptually simple to extend our modeling for this. E.g., we could carry out our analysis separately for each hour of the day (or another appropriate time duration).

[2] By temporal locality, we mean that over relatively short time-scales (a day to a few days), the key features tend to change little, whereas over longer time-scales (weeks to months), they might undergo more substantial changes.

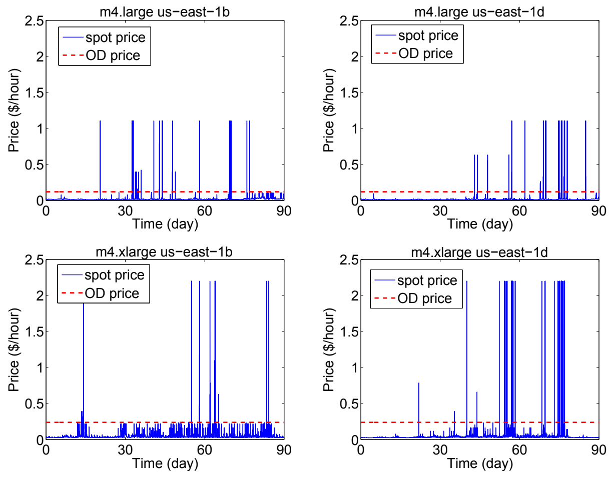

Figure 2: Sample spot price traces of m4.large and m4.xlarge in us-east-1b and us-east-1d during the 90-day period (2015/07/08-2015/10/06).

We present a small but representative subset of our overall results due to space limits. We vary the spot markets and bids and show the above assessment metrics in Table 2. We use spot price traces collected during a 90-day period in Figure 2, with bid b picked from $\{0.5d, d, 2d, 5d, 10d\}$, where d is the corresponding on-demand price. For purposes of comparison, we also present the above metrics based on predictions of $L^s(b)$ and $\bar{p}^s(b)$ by using the empirical CDF of historical spot prices within the same sliding window (updated dynamically). For this baseline, $\hat{L}^s(b) = H \cdot \text{Prob}(p^s_t \le b)$ wherein $H = 7$ days is the history window size, and $\hat{\bar{p}}^s(b) = \text{E}[p^s_t|p_t \le b]$. Under most (market, bid) pairs, f and ξ are below 10%, which we consider a reasonable demonstration of the efficacy of our predictors. We also find that f and ξ using our predictors are almost always better (lower) than those obtained under the CDF-based approach, which indicates that our models exhibit better temporal locality than the CDF of raw spot prices.

3.2 Usage: Combining Spot and On-Demand

Recall the resource wastage we identified for OD+Spot_Sep. In order to improve resource utilization and reduce costs, we devise a content placement scheme that mixes hot and cold content among on-demand and spot instances to achieve a desirable balance between using procured resources well and keeping spot failure induced performance degradation within tolerable limits. Figure 3 provides an illustration of the basic idea using a hypothetical heavy-tailed content popularity distribution.

In our model, we denote as α ($0 < \alpha \le 1$) the percentage out of the whole working set that needs to be held in memory. A choice of α equal to or close to 1 represents an in-memory data store, whereas a smaller α probably represents a caching tier of a multi-tier application where only a

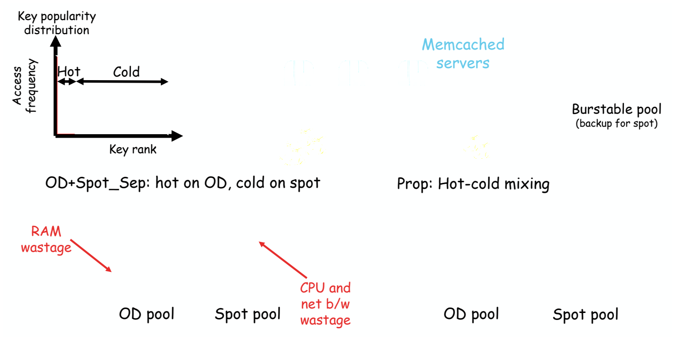

Figure 3: Illustration of hot-cold mixing and burstable-based passive backup of spot contents.

small subset of the working set needs to be in memory to improve performance as well as saving costs for the caching tier. We put the data into different popularity classes by denoting as H $(0 < H \leq \alpha)$ the portion of working set that is hot[3]. Both α and H are application/workload-specific and could also be time-varying.

Denote as B the set of all bid values. Denote as the set of all **markets** $S = \{\{\text{regions}\} \times \{\text{availability zones}\} \times \{\text{Instance types}\} \cup \{\text{On-demand}\}$. Note that on-demand instances can be viewed as a special type of spot instance for which the bid price equals the fixed price of the on-demand instance and the lifetime is infinite. Then we denote as $x_t^{sb}, y_t^{sb} \in [0, 1]$ the portions of hot and cold data that will be placed in market $s \in S$ using a bid $b \in B$ at time t, respectively. The amount of resource capacities allocated in (s, b) should be sufficient to accommodate the corresponding percentage of the working set and request arrival rate. We leave the details of how to use our models in an optimization problem (with the goal of minimizing costs) to solve for x_t^{sb}, y_t^{sb} to Section 4.

3.3 Recovery: Burstable-based Passive Backup

We build upon ideas from the classic primary-backup fault-tolerance technique [3]. Unlike the classic technique, however, we only replicate the hot cached elements that our hot-cold mixing has placed on spot instances (i.e., these are the cache elements we would like to be able to access with low latencies even upon the revocation of the spot instances holding them). It would be desirable to achieve low costs while keeping the failure recovery period short such that the performance degradation is kept small. Given that the amount of hot data is usually small, smaller instances might be a good fit for backup due to their low prices if they can still offer adequate capacities.

We illustrate different cases of how the failure recovery could be carried out in Figure 4. In case 1(a), we use the backup (“B”) to warm up (“copy”) the cold cache of the replacement (“R”), which finishes before the spot instance (“S”) gets revoked. Then we reconfigure the load balancer so that the requests are redirected to R. In case that R is not warmed up yet when S is revoked (1(b)), we could temporarily serve the requests and warm up R using B (events $4-6$), or the back-end database (events $6'-7'$). If R is not even started upon spot revocation (case 2), we could apply similar strategies as in case 1(b). Note that we only store hot data on B. For the warm-up process from B to R only the first request to the data item is served by B, and R is updated with the latest value in the background; all the subsequent requests for the same data item would be served by R. Therefore, even if the network latency between B and R becomes high, the overall performance could still be acceptable.

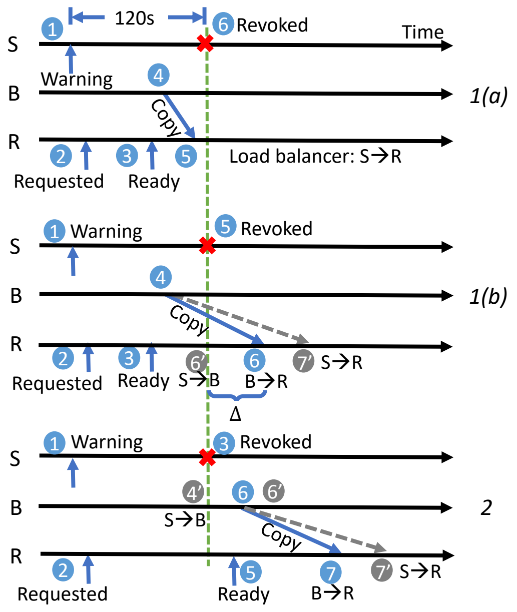

Figure 4: Different cases for the failure recovery process. “S, B, R” denote spot, backup and replacement, respectively. Events $6'-7'$: B is only used for warming up R. Events 4-7: B is used for both warming up R and serving requests.

The number of backup instances can be determined based on the amount of hot content and the choice of backup instance type. In order to expedite the warm-up process and mitigate the performance degradation during the interim Δ, it would be desirable if the backup has high CPU/network capacities per unit RAM. From Table 1, we notice that burstable instances, *if operated at their peak capacity*, may be a better option than others in that they offer superior CPU and network bandwidth per unit RAM for every dollar invested (Table 3). Although the resource capacities of burstables are variable, *they are not random*; in fact, based on EC2’s documentation and our measurements (Figure 5), the CPU capacity and network bandwidth follow deterministic token-bucket mechanisms which the tenant can control

[3] We consider as hot the most popular subset of the overall working set that accounts for 90% of accesses. There can be other ways of defining hotness and it is also possible to consider more levels of popularity than just two as we do. Our formulation easily extends to incorporate these.

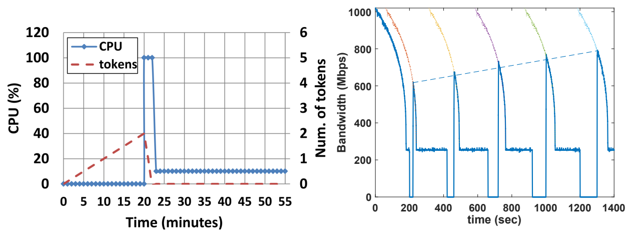

Figure 5: The token-bucket mechanisms of the CPU capacity and network bandwidth for t2.micro.

Burstable type	Unit price ($/hour)	OD price ($/hour)
t2.nano	0.0065	0.0425
t2.micro	0.013	0.0454
t2.small	0.026	0.0511
t2.medium	0.052	0.1022
t2.large	0.104	0.125

Table 3: Cost comparison of EC2 burstable instances. The OD price is calculated based on the peak capacity of burstable instances and the unit price of OD in Table 1.

by adjusting its resource usage patterns. In our implementation, we keep track of the token status of the burstables and adapt to different strategies based on how much data needs to be copied and how many tokens are available.

4. System Design and Implementation

4.1 Predictive Optimizer Design

We consider a slotted time system with time slots of length Δ (predictive control window, e.g., one hour) indexed by t. We cast an online optimization problem that takes predictions of workload properties - request arrival rate $\hat{\lambda}_t$ and "working set" size $\hat{M}_t$ as inputs, procures instances from multiple EC2 markets with appropriate bids, and devises a suitable cache placement. The goal of our formulation is to minimize the tenant's costs while satisfying performance target during the next time slot.

Denote as N_t^{sb} and $\tilde{N}_t^{sb}$ the number of existing instances ("system state") and additional instances to be procured ("control actions") at the start of time slot t, respectively, in market $s \in S$ with bid $b \in B$, with c^s and m^s being the number of vCPUs and amount of RAM [4] for the corresponding instance type. $\tilde{N}_t^{sb}$ could be negative (corresponding to a need for de-allocation).

[4] Although we also consider network bandwidth - crucial to memcached performance - in our allocation decision-making, we omit it and conduct our discussion only in terms of CPU capacity and RAM.

Denote as $\hat{M}_t$ the predicted working set size in time slot t. Based on our key idea on *hot/cold mixing*, we have

$$\begin{aligned}
\sum_{s\in S}\sum_{b\in B} x_t^{sb} &= H,\ \forall s\in S,\ b\in B \\
\sum_{s\in S}\sum_{b\in B} y_t^{sb} &= \alpha - H,\ \forall s\in S,\ b\in B \\
(N_t^{sb}+\tilde{N}_t^{sb})m^s &\geq (x_t^{sb}+y_t^{sb})\hat{M}_t,\ \forall s\in S,\ b\in B \\
\sum_{s\in S^{OD}} (x_t^{sb}+y_t^{sb}) &\geq \zeta
\end{aligned} \tag{1}$$

where the last constraint guarantees that at least ζ fraction of the entire working set would be placed on on-demand instances which preserve service availability (and performance to some extent) upon one or more bid failures. S^{OD} is the set of all on-demand instance types. Both α and H are application-specific and could also be time-varying.

Denote as l_t the latency target and l^{TGT} the target performance. Define $F(.)(\in [0,1])$ as the CDF of data popularity distribution (measured and updated in an online manner). Since $F(\alpha)$ percentage of the arrivals would be hit and the rest would be miss, we have

$$F(\alpha)l^{HIT} + (1-F(\alpha))(l^{HIT}+l^{MISS}) \leq l^{TGT}$$

where l^{HIT} and l^{MISS} represent the hit latency and the additional latency due to miss, respectively. l^{MISS} can be measured empirically and l^{HIT} depends on the resource allocation. Given l^{TGT} and α, we can obtain l^{HIT} as the latency bound of an instance with 100% hit rate in order to satisfy the overall performance target.

Define $l_t = \phi(\lambda_t, \text{vCPU}, \text{RAM})$ to capture how resource allocation affects application performance wherein λ_t is the request arrival rate. The $\phi(.)$ function can be either empirically measured offline and updated periodically online (e.g., $\phi(.)$ could be a regression model [23] or a lookup table based on the performance profiling results as is done in our evaluation), or theoretically modeled, e.g., via queuing analysis [10, 43]. Given the latency bound l^{HIT}, we have ($\forall s \in S, b \in B$)

$$\begin{aligned}
\lambda_t^{sb} &= \hat{\lambda}_t\left(\frac{x_t^{sb}}{H}F(H) + \frac{y_t^{sb}}{\alpha - H}(F(\alpha)-F(H))\right) \\
\phi(\lambda_t^{sb}, c^s(N_t^{sb}&+\tilde{N}_t^{sb}), m^s(N_t^{sb}+\tilde{N}_t^{sb})) \leq l^{HIT}
\end{aligned}$$

wherein λ_t^{sb} represents the arrival rates assigned to instances in market s under bid b. According to Eq. 1, we would allocate enough RAM capacity so that the full α percentage of dataset could be in memory. Therefore, we can re-write the above equation as follows:

$$(N_t^{sb}+\tilde{N}_t^{sb})\lambda^{sb} \geq \lambda_t^{sb},\ \ \forall s\in S, b\in B \tag{2}$$

wherein λ^{sb} is the maximum workload on one instance in market s under bid b without violating the latency bound

l^{HIT}, when the full dataset $(x_t^{sb}+y_t^{sb})\hat{M}_t$ is held in memory. λ^{sb} can be obtained via offline performance profiling. This simplification converts Eq. 2 into a linear constraint.

Resource costs: Denote as $\hat{\bar{p}}_t^{sb}$ the predicted average price during t ($\hat{\bar{p}}_t^{sb}$ is the on-demand price for on-demand instances). Therefore, the total resource costs (with a slight over-estimation due to ignoring bid failures during this time slot Δ) can be expressed as $\sum_{s\in S}\sum_{b\in B}\hat{\bar{p}}_t^{sb}(N_t^{sb}+\tilde{N}_t^{sb})\Delta$.

Bid failure penalty: When spot price exceeds the bid associated with an instance, the tenant may incur one or both of (i) performance degradation due to the capacity loss and (ii) explicit additional costs incurred by remedial actions it may take. Different choices for (ii) have been explored for EC2 spot instances wherein a warning is offered a little ahead (2 minutes as of this paper) of spot revocation. In effect, procuring a spot instance is made "more expensive" by an amount that is a function both of the spot price evolution as well as the bid. Whereas this additional expense is explicit for (ii), it has to be translated from performance degradation for (i) in a tenant-specific way. We devise the following simple (yet effective as seen in our evaluation) model for capturing this as a "penalty" or "loss rate" associated with a (spot instance, bid) combination given as $\frac{\beta_1 x_t^{sb}+\beta_2 y_t^{sb}}{\hat{L}^s(b)}$ wherein $\hat{L}^s(b)$ denotes the predicted $L(b)$ in market s. β_1 and β_2 are coefficients reflecting the penalty of losing hot data and cold data, respectively. The choice of coefficients would be tenant/application-specific. In our evaluation, we choose the values such that all terms/parameters in the tenant's cost function are non-negligible.

Resource deallocation penalty: De-allocating resources may not be "free". CPU is a relatively "stateless" resource that can be scaled up/down instantaneously without affecting application performance much (barring hardware caching/TLB-related effects), if fine-grained scaling is enabled. However, memory, even if scaled down assuming perfect prediction of the application's needs, might degrade application performance since the evicted data might become popular in future time slots. Therefore, we introduce into our cost model an additional term $\eta\max\{0,-\tilde{N}_t^{sb}\}$ in order to dampen memory de-allocation.

Optimization problem formulation: At the beginning of time slot t (proactive control window), we solve the following online control problem:

$$\text{Minimize}_{x_t^{sb},y_t^{sb}} \sum_{s\in S}\sum_{b\in B}\Bigg(\hat{\bar{p}}_t^{sb}(N_t^{sb}+\tilde{N}_t^{sb})\Delta + \eta\max\{0,-\tilde{N}_t^{sb}\} + \Delta\frac{(\beta_1 x_t^{sb}+\beta_2 y_t^{sb})\hat{M}_t}{\hat{L}^s(b)}\Bigg)$$

$$\text{s.t.}\quad (1),(2).$$

wherein N_t^{sb} and $\tilde{N}_t^{sb}$ are integer variables. Note that $\hat{\lambda}_t$ and $\hat{M}_t$ need to be predicted via appropriate predictive models, e.g., an AR(2) model $\hat{\lambda}_t=\gamma_1\lambda_{t-1}+\gamma_2\lambda_{t-2}$, before solving the optimization problem.

4.2 Implementation

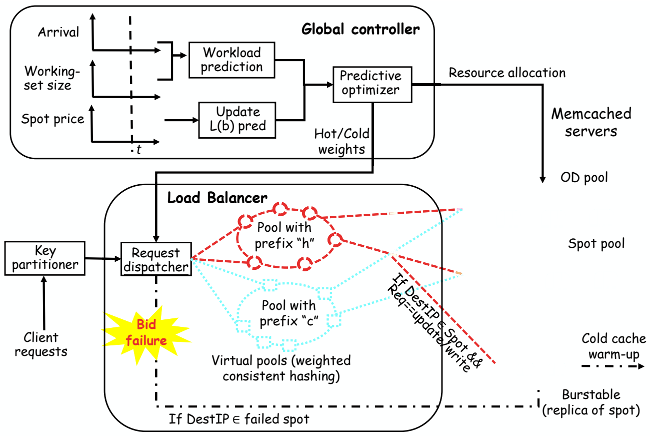

Figure 6: Key elements of our prototype implementation.

Figure 6 depicts the salient aspects of our prototype implementation on EC2. We use Facebook's mcrouter [25] as the front-end of our memcached cluster wherein we implement these two components of our overall system: (i) a key partitioner, and (ii) a load balancer. A final component of our system is a global controller. Next, we describe these 3 components in detail (including a discussion of some elements for which we assume existing work can be leveraged).

Key partitioner: We create Bloom filters using access frequency-based heuristics which are periodically refreshed to keep track of "hot" keys (which we define as keys with access frequency above a certain threshold within a given time window). More sophisticated heuristics such as ghost lists [26], probabilistic counters [9], Count-Min Sketch [7], etc., can easily replace our simpler approach. This information is used to annotate keys as hot (via a prefix "h") or cold (via a prefix "c"). This can be easily generalized to additional popularity levels if needed. If certain cold data becomes hot, the key partitioner would detect this change via well-studied techniques in the prior work [5, 17] and re-assign prefixes to the corresponding keys.

Load balancer: Periodically, mcrouter updates its records of the hot and cold weights of all memcached instances based on the output of the optimization problem solved by the global controller. We leverage mcrouter's PrefixRouting technique ("h" and "c" in our case) to create separate "virtual" pools for hot and cold keys. These pools exist on the same set of memcached servers, allowing hot/cold key segregation without having to resort to instance separation. Within each virtual pool, we use a weighted consistent hashing algorithm (implemented in mcrouter) to forward the requests such that the amount of hot/cold data placed on the memcached nodes are proportional to their hot/cold weights.

For approaches with passive backup, during normal operations, update/write requests for the content on spot instances are also sent to the appropriate backup nodes to maintain consistency. Read requests are not served by backup nodes based on burstable instances, allowing them to accumulate CPU and network bandwidth tokens in anticipation of the work they would need to do upon a spot revocation. Finally, upon receiving warning about an impending spot revocation, we allocate a new on-demand instance as its replacement. We implement all the options - direct transition to the replacement or via a backup node - discussed in Section 3.3 (recall Figure 4) to be able to compare them in our evaluation.

Global controller: The global controller, running on dedicated instance, periodically updates workload properties and spot price predictions, solves the online optimization problem from Section 4 and determines the hot x_t^{sb} and cold y_t^{sb} weights for the market s with bid b. This information is then sent to the load balancer (implemented within mcrouter)[5]. Within the same market and under the same bid, the weights are evenly distributed among all instances, i.e., each instance obtains the same portions of hot and cold weights. For a multi-tier application, the global controller would also be responsible for coordinating resource scaling for/across different tiers, e.g., scaling up the CPU capacity of the database tier vs. scaling up the RAM capacity of the caching tier. We do not consider these issues in our design [11, 44]; in our evaluation we locate our back-end on an instance provisioned for worst-case workload needs. Finally, we also implement a reactive element in our global controller to take corrective resource allocation decisions in case of unexpected events such as flash crowds that are not captured by the above predictive mechanisms. We do not discuss it due to space constraints. The idea of combining complementary predictive and reactive elements into a hierarchical control framework to deal with unexpected flash crowds and misprediction of the workloads is a very well-established design principle numerous specific forms of which have been studied [10, 33, 37, 44].

5. Experimental Evaluation

5.1 Experimental Setup

Workloads: We scale the Wikipedia access trace [42] to create workloads with different peak arrival rates and maximum working set sizes. We use the YCSB benchmark to generate requests (100% read) to memcached instances based on Zipfian popularity distribution with the Zipfian parameter varied in the range 0.5-2 (higher value yields a more skewed popularity distribution).

memcached **configuration:** For the prototype experiments, we set the dynamic working set size to 25-60GB (with 4KB item size) and the maximum arrival rate to 320k operations per second (Ops), respectively. We consider a average latency target of 800us and an 95%ile latency target of 1ms. Our workloads can be handled by a single mcrouter running on a well-provisioned c4.2XL on-demand instance and we do not need to consider replicating mcrouter.

[5] In this work, we assume a reliable mechanism to commit this information consistently across all mcrouters if there exist more than one load balancers. Systems such as Chubby [4], Zookeeper [15] can be used for this purpose.

Approach	Uses our spot modeling?	Uses our hot-cold mixing?	Passive backup?
ODOnly	×	×	×
OD+Spot_Sep	√	×	×
OD+Spot_CDF	×	√	×
Prop_NoBackup	√	√	×
Prop	√	√	√

Table 4: Procurement approaches that we compare.

EC2 instance types considered: Although our implementation is flexible enough to consider the full array of EC2 instance on-demand, spot, and burstable types, we experiment using a limited set both for ease of understanding/explaining interesting phenomena and for addressing certain peculiarities of memcached. It is well-known that memcached does not scale well beyond four cores [6]. Therefore, we exclude larger instances and only consider instance types within the m3.*, c3.* and r3.* series with less than or equal to four vCPUs as candidates for on-demand (a total of 6 instance types). For spot instances, we consider the 90-day long spot price traces of m4.large and m4.xlarge collected from the us-east-1c and us-east-1d regions (Figure 2). We use the first 7-day sub-trace of each spot price trace as the training data to obtain initial parameters of our spot feature predictors (recall Section 3.2). We pick d and $5d$ as the bid prices for each spot market wherein d is the on-demand price for that spot market. All our experiments are on instances running Ubuntu 14.04 operating system.

Procurement approaches: We consider two operating modes of our system: one without a passive backup (Prop_NoBackup) and the other with a passive backup (Prop). In Table 4, we list these approaches as well as the baselines presented in Section 2.3. We offer a summary comparison of all approaches in terms of whether each: (i) uses our spot modeling and prediction, (ii) exploits hot-cold mixing, and (iii) employs a passive backup.

Note on the scope of our evaluation: Our techniques are useful if workloads exhibit read-heavy accesses skewed popularity, which are found to occur frequently as suggested in prior work [1, 42].

5.2 Efficacy of Our Spot Feature Modeling

We compare the following two approaches: OD+Spot_CDF and Prop_NoBackup. The only difference between these approaches is how spot instance features (residual lifetime and average spot price during lifetime) are predicted.

First, we conduct trace-driven simulations and compare long-term costs and memcached performance with the two

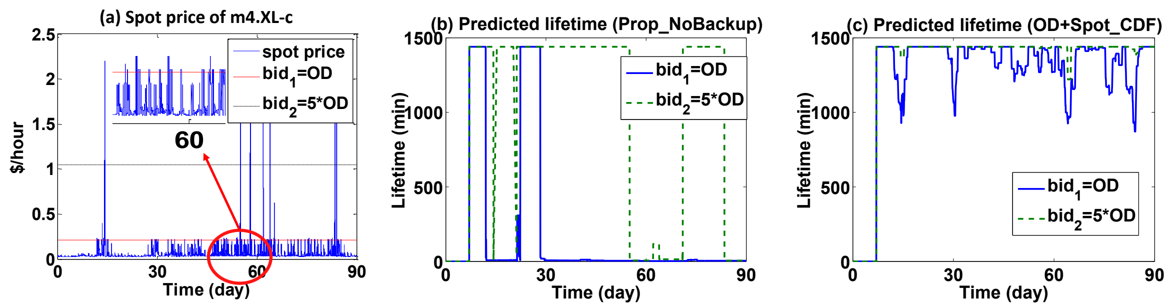

Figure 8: Spot price and predicted instance residual lifetime under different strategies.

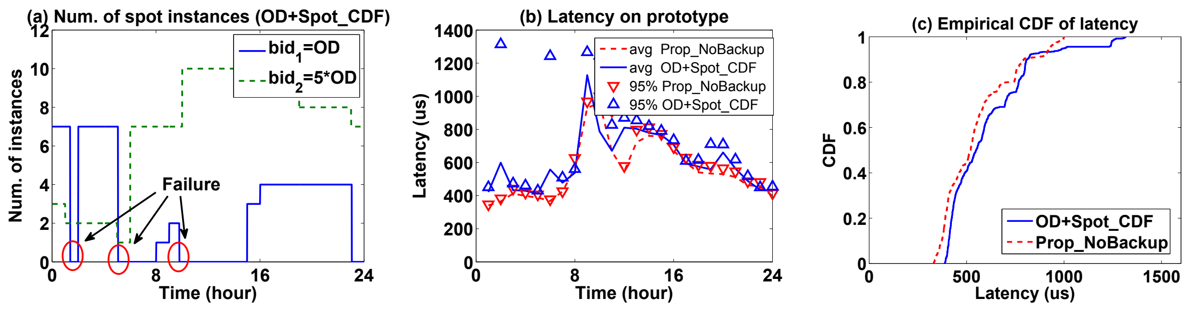

Figure 9: Resource allocation and performance measurement on EC2 (impact of spot prediction).

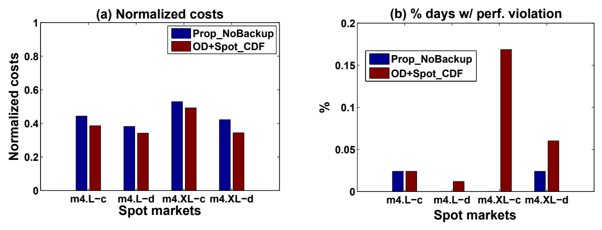

Figure 7: Normalized costs (divided by costs of ODOnly) and percentage of days when the performance target is violated when using Prop_NoBackup and OD+Spot_CDF.

approaches assuming only one of the four spot markets of Figure 2 is available at a time. This corresponds to the real-world scenario where the tenant is forced to stick to a single market for performance/management concerns, e.g., the tenant wanting to place its cache nodes in physical proximity of its back-end or its client base. We specify a peak arrival rate to 500k Ops and the maximum size of the cache contents (henceforth also called the working set size) to 100 GB. The popularity distribution is zipfian with parameter 2.0. Figure 7 shows the normalized costs (obtained by dividing by the costs for ODOnly) and percentage of days when the performance target is likely to have been violated (more than 1% of requests are affected by bid failures). We find that Prop_NoBackup is able to achieve much better performance than OD+Spot_CDF while still obtaining comparable cost-savings (only 5% lower than). We may attribute these improvements to our spot feature modeling techniques. Figure 8 shows the predicted instance residual lifetimes available to the two approaches when operating in the spot market m4.XL-c (OD+Spot_CDF performs the worst in this market), thereby helping us appreciate the role of our spot feature modeling. By directly modeling and predicting residual lifetime instead of relying on cumulative distribution of historical spot prices, our predictor is able to help the optimizer avoid using bid$_1$ (lower bid) when the spot price exceeds this bid frequently (e.g., the interval between the 30-th and 60-th days) whereas the CDF-based approach fails to do so.

We run 24-hour long experiments (with spot prices for the 51-st day for the market m4.XL-c) on our EC2-based prototype. With OD+Spot_CDF, our tenant experiences three partial bid failures (a subset of the procured spot instances fail) whereas with Prop_NoBackup there are no such failures. We show our performance measurements in Figure 9. We find that Prop_NoBackup outperforms OD+Spot_CDF. In particular, although the two approaches offer similar average latency, Prop_NoBackup has a superior tail latency owing to fewer spot revocations.

5.3 Efficacy of Our Hot-Cold Mixing

We now compare Prop_NoBackup with OD+Spot_Sep which are identical in all respects except that the former uses hot-cold mixing whereas the latter places all hot content on on-demand instances and all cold content on spot instances. We construct experiments using spot prices from market m4.L-d. We run 24-hour experiments on (for day 45) our EC2-based prototype. We specify a maximum arrival

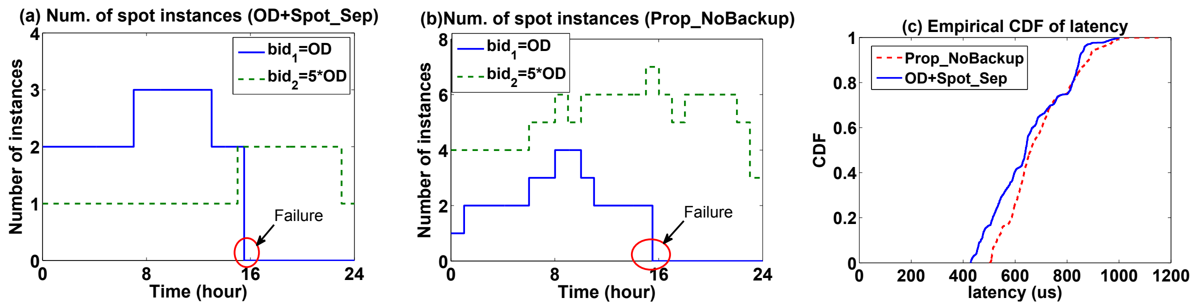

Figure 10: Instance allocation and performance measurements in spot market m4.L-d (impact of hot/cold mixing).

rate of 320 kops and a maximum working set size of 60 GB. Figure 10 shows the resource allocation and performance measurements in the spot market under different strategies. First, we find that, our spot feature modeling enables the placement of multiple bids for both strategies, thereby causing only a subset of spot instances fail (at any given time) over the course of the day. In particular, we find that Prop_NoBackup allocates fewer spot instances using the lower bid (bid_1) than using the higher bid (bid_2) - our predictive optimizer deems it risky to allocate all spot instances under the lower bid (despite the possibly lower costs with the lower bid). After the bid failure, we find that the number of spot instances with the lower bid reduces to 0 under both strategies, since our conservative prediction of VM residual lifetime is updated to a much lower value.

Second, as shown in Figure 10(c), we observe that, on average the two approaches offer comparable performance. Prop_NoBackup occasionally exhibits poorer tail latency. This may arise from the higher likelihood of resources reaching saturation levels with Prop_NoBackup due to its more aggressive resource usage. In addition, Prop_NoBackup offers correspondingly better costs - around 20-95% extra-savings over OD+Spot_Sep based on our results from Section 5.5. This illustrates the cost vs. performance trade-off associated with our hot-cold mixing and also serves to set the context for exploring how adding a passive backup can help alleviate this poorer performance.

5.4 Efficacy of Our Passive Backup

Recall that EC2 issues a warning 2 minutes prior to an impending spot instance revocation. We consider two different scenarios, (A) wherein the replacement instances requested upon receiving such a warning become operational before the actual revocation occurs and (B) wherein they become operational only past the actual revocation. (A) and (B) correspond to cases 1(b) and 2 in Figure 4.

Scenario (A): Existing measurement studies (including our own) show that it usually takes about 100 seconds to launch a small/medium-sized on-demand instance [8, 22]. Therefore, this (more desirable) scenario is likely to be the usual (or at least frequent) case. We consider Prop employing the following backup options: (i) t2.medium, (ii) m3.medium and (iii) c3.large. (i) is a burstable instance whereas (ii) and (iii) are regular on-demand instances that are closest in their RAM capacities to t2.medium. Our workload has 40k Ops maximum arrival rate and 10GB working set size out which 3GB we deem as hot data, with the zipfian parameter for popularity distribution chosen to be 1. In Figure 11(a) we compare latency during the recovery period for these 3 configurations of Prop versus those for Prop_NoBackup (no backup, so all misses serviced from a slow back-end) and OD+Spot_Sep (only cold data lost upon revocation). We focus only on the content affected by a bid failure. $t = 0$ corresponds to the beginning of the period when a newly launched replacement is ready and the copying process begins. The copying finishes at round $t = 300$ (latency falls back to within 1.05x the target average latency). The backup is not used for serving the requests during this process. As expected, warm-up using passive backup provides much better performance than Prop_NoBackup in terms of both the maximum latency and the time for it to settle down to pre-revocation levels (length of the warmup period). The burstable t2.medium is able to offer similar recovery-time performance as the about 2x expensive c3.large, and outperforms the slightly expensive m3.medium, while getting closer to OD+Spot_Sep where no loss of hot data. In addition, by using the burstable instance t2.medium, the 95% latency during failure recovery improves by 25% compared to a backup based on regular instance m3.medium.

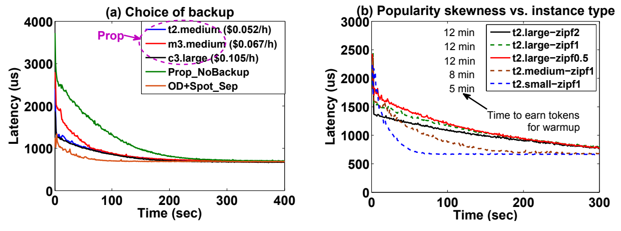

Figure 11: Recovery latency under (a) different choices of backup and (b) different content popularity distributions and instance types. The curves for t2.medium and c3.large in (a) almost fully overlap.

Figure 11(b) shows the warmup time under different popularity distributions and burstable instance types. The dataset sizes are chosen to be closest to their RAM capacities. We also show the times needed to earn enough credits for the burstable instances to burst during the warmup period, which could reflect the mean time between failures that the burstables might afford for failure recovery. For the same instance type, we observe longer warmup times when the popularity distribution is less skewed, since it would take less time to copy over the hottest keys (enough to bring the latency back to normal) under more highly skewed distributions.

Scenario (B): Here, the replacement for the revoked spot instance is not yet ready for use when the actual revocation occurs. During this interim period, the backup may both assist in warming up the replacement and also serve content for the revoked spot instances. We emulate such scenario on our prototype and still observe similar performance improvement over Prop_NoBackup when the interim period is not too long such that the burstables use all resource tokens. Due to space limit, we omit these results here.

5.5 Impact of Workload Properties on Long-Term Costs

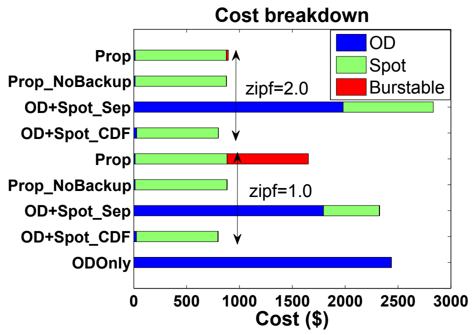

Figure 12: Long term (90-day) cost breakdown (max. arrival rate: 500kops; max. working set size: 100 GB).

Finally, we focus on understanding cost savings that Prop can offer relative to other approaches. We are also interested in exploring how these savings depend on important workload properties (arrival rate, working set size, and popularity distribution). Towards this, we create a variety of workloads by picking different values for the maximum arrival rate $\in \{100\text{kops}, 500\text{kops}, 1000\text{kops}\}$, the maximum working set size $\in \{10\text{ GB}, 100\text{ GB}, 500\text{ GB}\}$, and the Zipfian parameter $\in \{1.0, 2.0\}$, which yields 18 different workloads in total. In each experiment, we use all the on-demand/spot types and all spot markets with three-month price traces described in our experimental setup. Figures 12 and 13 show the long term cost breakdown and overall costs for various approaches.

We make several observations from these experiment results. First, as expected, Prop_NoBackup is able to outperform OD+Spot_Sep and ODOnly under all workload scenarios. It matches the costs for OD+Spot_CDF, while offering better tail latency due to incurring fewer spot revocations (recall from Section 5.2). Second, using OD+Spot_Sep may surprisingly result in costs even higher than ODOnly when the workload is highly skewed (e.g., Zipf=2.0). This is because when the popularity distribution is highly skewed (only a very small subset of the data is "very hot") OD+Spot_Sep may end up wasting too much RAM capacity on on-demand instances and CPU/network capacity on spot instances. Third, for certain workloads with less skewed popularity distribution, the cost of passive backup may not be negligible and the tenant must carefully consider its pros and cons. In Figure 13, the normalized cost of Prop approaches 1 for such workloads. However, as the workload becomes more skewed (Zipf=2.0), the cost of the backup becomes negligible. This can also be verified from Figure 12. Fourth, when we fix the maximum working set size, the maximum arrival rate has little impact on the normalized costs (although it does affect the absolute costs); however, when we fix the maximum arrival rate, the maximum working set size influences the normalized costs a lot. Additionally, workloads with higher ratios of arrival rate vs. working set size may benefit more from Prop_NoBackup than workloads with lower ratios. In the former scenarios CPU capacity becomes the dominant resource; since the unit price for vCPU is much higher than that of RAM (recall Table 1), the cost savings would be higher for such workloads if Prop_NoBackup were able to find a suitable combination of on-demand and spot instances that reduces wastage.

6. Related Work

Spot instance for tenant procurement: A large body of recent work has explored the use of spot instances (possibly in combination with regular instances) for cost-effective operation of a variety of workloads, e.g., delay-tolerance batch jobs [19, 30, 38, 39, 41, 51], video streaming [13], and (closer to our focus) for in-memory storage [50]. Various techniques have been explored for spot price modeling and prediction, e.g., , auto-regressive models [2, 45, 53] or empirically measured probability distributions of key parameters [16, 24, 35, 36, 39, 40] (generally described as "CDF-based" by us), to more complex Markovian models [30, 38, 51]. Simple regressive models might fail to provide insights on how the spot price might evolve in the long run since it could change at a minute's granularity. CDF-based approaches, although offering an improved treatment of longer term properties than regression models, usually discard valuable temporal information about the continuity of the spot price staying below different bid values. Therefore, they may fail to capture well the continuity of service availability, which is of great concern particularly for long-lived and "stateful" applications. To our knowledge, one exception on this front is [38] wherein the "sojourn time" of a spot instance procured via a particular bid is modeled and predicted via a semi-Markov chain. However, tenant control based on such multidimensional models would likely suffer scalability limitations given the large number of spot markets and bid space - our approaches are geared towards

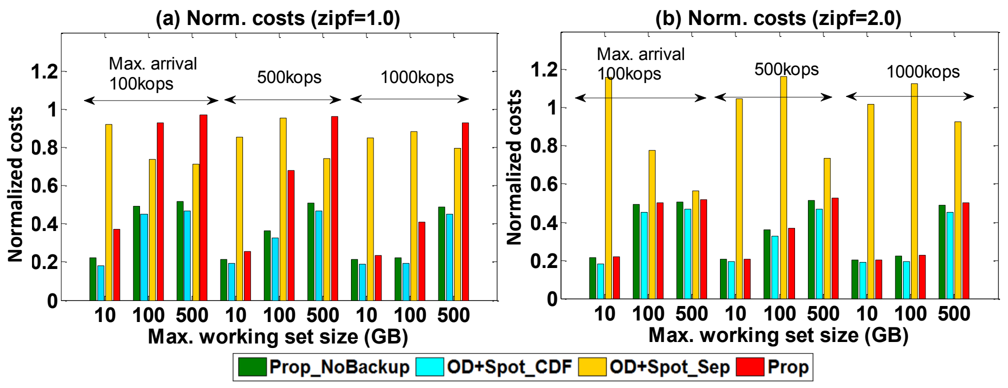

Figure 13: Impact of workload properties on long term costs.

achieving a balance between these scalability concerns (for effective use in online control) and accuracy.

Burstable instances for tenant procurement: These instances have not received much attention, possibly due to their recent introduction (ca. 2014 for the t2 family). A few papers [20, 47] and technical blogs [18, 31] have benchmarked their performance by comparing their capacity variability against regular instances. Close to our work, [47] focuses on "staggering" burstable instances (in time) for batch jobs by intentionally injecting delays between successive jobs. Our work offers another novel and compelling use case for these instances as passive backups for in-memory caches. A key distinguishing aspect of our work is that we exploit the *deterministic* nature of CPU and network bandwidth variations which are regulated by token buckets (rather than viewing them as externally-controlled random phenomena).

Operating in-memory caches on public clouds: There has been significant work on deploying in-memory data caches on public clouds with a variety of foci, e.g., improving application performance by addressing load imbalance [6, 14, 52], synchronization/concurrency [12], geo-replication [34, 49, 50], scalability [21, 32, 48], etc. These are all complementary to our work. Closest to our work, [50] blends spot and on-demand instances to lower costs for in-memory storage. This work considers a design space different from ours. Some of their key ideas include (i) using spot instances for read-only content and using on-demand instances for applying updates, and (ii) using replication across multiple spot markets for performance and fault tolerance. Our spot price prediction techniques might offer additional improvements were they to replace their CDF-based approaches. Similarly, it appears feasible to combine our ideas related to hot-cold mixing and use of burstable instances with theirs. On the other hand, our evaluation was restricted to a single-site setting whereas geo-replication receives significant attention in their work. Similarly, their ideas for using a small number of on-demand instances (high availability) to speedily serve write requests may be useful in extending the scope of our work from its current focus on read-heavy workloads. Overall, we consider our papers to be highly complementary.

7. Conclusion

We studied cost-effective operation of dynamic memcached workloads on Amazon EC2 by combining regular instances with spot and burstable types. We proposed novel ideas that we believed would offer improvements over the state-of-the-art or complement it: (i) scalable data-driven spot price prediction, (ii) intelligent hot-cold mixing among on-demand and spot, and (iii) burstables for fast/cheap passive backup. We implemented these ideas into a prototype system on EC2. Our experimental evaluation suggested significant cost-savings while offering comparable or better performance than baselines representing conventional wisdom. We believe that other cloud providers are also likely to offer similar cheap instances studied in this paper as it is a natural way to increase their market-base, which would make our approach serve as a more general solution to public cloud in-memory cache applications.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ada Gavrilovska for their valuable feedback that helped us improve the quality of our paper. Urgaonkar was supported, in part, by the NSF CAREER award 0953541 and an IBM faculty award. Kesidis was supported, in part, by a DARPA XD3 grant, NSF NeTS grant 1526133 and a Cisco Systems URP gift.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. ACM SIGMETRICS'12*, 2012.

[2] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. In *Proc. CloudCom*, 2011.

[3] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In *Distributed systems (2nd Ed.)*,

pages 199–216. ACM Press/Addison-Wesley Publishing Co., 1993.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. USENIX OSDI*, 2006.

[5] L. CHEN, H. TANG, X. LUO, Y. BAI, and Z. ZHANG. Gain-aware caching scheme based on popularity monitoring in information-centric networking. *IEICE Transactions on Communications*, 2016.

[6] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. EuroSys*, 2015.

[7] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1), 2005.

[8] Ec2 boot time, 2016. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html`.

[9] P. Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1), 1985.

[10] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14, 2012.

[11] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch. Softscale: stealing opportunistically for transient scaling. In *Proc. Middleware*, 2012.

[12] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey. Mercury: Bringing efficiency to key-value stores. In *Proc. SYSTOR*, 2013.

[13] J. He, Y. Wen, J. Huang, and D. Wu. On the cost–QoE tradeoff for cloud-based video streaming under Amazon EC2's pricing models. *Circuits and Systems for Video Technology, IEEE Transactions on*, 2014.

[14] Y-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proc. ACM SOCC*, 2013.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, 2010.

[16] B. Javadi, R. Thulasiramy, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Proc. IEEE UCC*, 2011.

[17] S. Kang, S. Lee, and Y. Ko. A recent popularity based dynamic cache management for content centric networking. In *Proc. ICUFN'12*, 2012.

[18] Patrick Kennedy. Testing new aws t2 instances with linux-bench benchmark. `https://www.servethehome.com/testing-aws-t2-instances-linuxbench-benchmark/`, 2014.

[19] S. Khatua and N. Mukherjee. Application-Centric resource provisioning for amazon EC2 spot instances. In *Euro-Par Parallel Processing*. Springer, 2013.

[20] P. Leitner and J. Scheuner. Bursting with Possibilities–An empirical study of credit-based bursting cloud instance types. In *Proc. IEEE/ACM UCC*, 2015.

[21] S. Madappa. Ephemeral volatile caching in cloud. `http://techblog.netflix.com/2012/01/ephemeral-volatile-caching-in-cloud.html`, 2012.

[22] M. Mao and M. Humphrey. A performance ctudy on the vm startup time in the cloud. In *Proc. IEEE CLOUD*, 2012.

[23] A. Marathe, B. Harris, D. K. Lowenthal, B. R. de Supinski, B Rountree, M. Schulz, and X. Yuan. A comparative study of high-performance computing on the cloud. In *Proc. HPDC'13*, 2013.

[24] M. Mattess, C. Vecchiola, and R. Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In *Proc. IEEE HPCC*, 2010.

[25] Facebook mcrouter. `https://github.com/facebook/mcrouter`.

[26] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. USENIX FAST*, 2003.

[27] Memcached, 2016. `https://memcached.org/`.

[28] Memcached cloud. `https://redislabs.com/memcached-cloud`, 2016.

[29] Memcachier. `https://www.memcachier.com/`, 2016.

[30] I. Menache, O. Shamir, and N. Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *Proc. IEEE ICAC*, 2014.

[31] A. Nhem. Cloudability. `https://blog.cloudability.com/how-cost-efficient-is-the-new-burstable-aws-t2-large/`, 2016.

[32] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and Ryan R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4), 2010.

[33] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "power" struggles: Coordinated multi-level power management for the data center. In *Proc. ACM ASPLOS*, 2008.

[34] P. N. Shankaranarayanan, A. Sivakumar, S. Rao, and M. Tawarmalani. Performance sensitive replication in geo-distributed cloud datastores. In *Proc. IEEE/IFIP DSN*, 2014.

[35] P. Sharma, D. Irwin, and P. Shenoy. How not to bid the cloud. In *USENIX Hotcloud*, 2016.

[36] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proc. Eurosys*, 2015.

[37] Z. Shen, S. Subbiah, X. X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. ACM SOCC*, 2011.

[38] Y. Song, M. Zafer, and K. Lee. Optimal bidding in spot instance market. In *Proc. of IEEE INFOCOM*, 2012.

[39] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. Spoton: A batch computing service for the spot market. In *Proc. ACM SOCC*, 2015.

[40] S. Subramanya, A. Rizk, and D. Irwin. Cloud spot markets are not sustainable: The case for transient guarantees. In *Proc. USENIX Hotcloud*, 2016.

[41] S. Tang, J. Yuan, and X-Y. Li. Towards optimal bidding strategy for amazon ec2 cloud spot instance. In *Proc. IEEE CLOUD*, 2012.

[42] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[43] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. ACM SIGMETRICS*, 2005.

[44] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM TAAS*, 3(1), 2008.

[45] R. M. Wallace, V. Turchenko, M. Sheikhalishahi, I. Turchenko, V. Shults, J. L. Vazquez-Poletti, and L. Grandinetti. Applications of neural-based spot market prediction for cloud computing. In *Proc. IDAACS*, 2013.

[46] W. Wang, B. Li, and B. Liang. To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds. In *Proc. USENIX ICAC*, 2013.

[47] J. Wen, L. Lu, G. Casale, and E. Smirni. Less can be more: Micro-managing vms in amazon EC2. In *Proc. IEEE CLOUD*, 2015.

[48] A. Wiggins and J. Langston. Enhancing the scalability of memcached. *Intel document, unpublished, http://software. intel. com/en-us/articles/enhancing-the-scalability-of-memcached*, 2012.

[49] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. ACM SOSP*, 2013.

[50] Z. Xu, C. Stewart, N. Deng, and X. Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *Proc. IEEE INFOCOM*, 2016.

[51] M. Zafer, Y. Song, and K. Lee. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *Proc. IEEE CLOUD*, 2012.

[52] W. Zhang, T. Wood, and J. Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *Proc. IEEE ICAC*, 2016.

[53] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang. Optimal resource rental planning for elastic applications in cloud market. In *Proc. IEEE IPDPS*, 2012.