Navigating the Public Cloud Labyrinth: A Study of Price, Capacity, and Scaling Granularity Trade-offs

Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta†, Qianlin Liang
The Pennsylvania State University, †IBM Research Almaden
Technical Report CSE-16-002

Abstract

Public clouds offer numerous virtual machine (VM) types with complex trade-offs that tenants must navigate for cost-effective resource procurement. We identify three key qualitative axes using which these trade-offs can be succinctly expressed: price dynamism, offered capacity dynamism, and granularity of resource scaling. We offer several examples of contemporary VM offerings and where they fit within our scheme. Using our classification, we argue that related work lacks techniques for effectively navigating this trade-off space. We attempt to fill this gap by devising modeling and control techniques that combine Amazon EC2 style spot and on-demand instances with fine-grained resource allocation knobs that have begun to be offered by public clouds. We implement a Memcached prototype tenant that procures VMs from Amazon EC2 using our techniques. Using extensive evaluation based on real-world workload and spot price traces, we demonstrate the efficacy of our approach (e.g., 40% cost savings compared to a baseline while maintaining performance goals).

I. INTRODUCTION

Public clouds offer a bewildering variety of virtual machine (VM or “instance”) procurement options to cater to the needs of a growing and diverse body of tenant workloads. E.g., Amazon EC2, Google Compute Engine, and Microsoft Azure all offer dozens of different types of VM configurations. Not surprisingly, tenant-side resource procurement has emerged as an important problem area both in research [45], [34], [18], [48], [42], [43], [10], [6], [20], [41], [5], [26] and practice (including startups and commercial products [9]). Tenant-side resource procurement must navigate the (often tricky and delicate) pros and cons associated with different VM types to reduce operational expenses while ensuring satisfactory workload performance.

Moving beyond the most natural (and likely the most well-studied) trade-off between the price of a VM and its advertised capacity, a tenant is still confronted with a dizzying array of differences and peculiarities. To ease reasoning about this complexity of public cloud interfaces, we employ a first-cut simplification, wherein we boil down the trade-offs among diverse VM types into three basic dimensions. Table I (top) presents a qualitative comparison of some well-known VM types from prominent providers along these three dimensions.

- **Price Dynamism:** Most VMs today have relatively static prices (change over months). Amazon EC2’s spot instances are the most prominent examples of VMs with dynamic pricing. Associated with a spot instance is a highly dynamic (may change in a matter of minutes) spot price that the cloud provider uses to incentivize certain types of tenant behavior (presumably to fix supply-demand mismatches and improve its own cost-efficacy). In Table I, we label Amazon EC2 spot instances as VM types that exhibit relatively higher dynamism in their prices compared to the other examples. Several papers have explored alternate dynamic pricing designs and tenant procurement in the context of such pricing [28], [13], [23], [44], [22], [30], [39], [45], [43]. It is reasonable to assume, therefore, that one or more forms of dynamic pricing will
continue to exist in the public cloud and continue to appeal to certain tenants. These alternate proposals for dynamic pricing would occupy different points on the dynamic pricing axis of Table I.

- **Offered Capacity Dynamism**: A key feature of public cloud VMs that is now well-appreciated by many tenants is that there may be a discernible gap between their advertised and offered capacities. This gap tends to be higher for cheaper VMs and likely arises due to differential cost-efficacy measures employed by the provider such as different degrees of consolidation and other types of resource under-provisioning for different VM types. Furthermore, numerous sources report that this gap can exhibit temporal variation, introducing additional complexity for tenants that wish to use such VMs [21], [40]. We find it useful to think of Amazon EC2 spot instances as representing an extreme form of such capacity dynamism - when a tenant’s bid for a VM fails, the VM offers “0 capacity.” Google’s pre-emptible VMs may be viewed similarly, although their prices are fixed and the tenant’s control on pre-emption is less direct (there is no bidding). Correspondingly, in Table I, we mark these VM types as offering high capacity dynamism. Capacity dynamism may be less explicit than for these types, e.g., as with Amazon EC2 burstable instances. Finally, plenty of evidence exists, including our own measurements [40], that such dynamism exists even for many VMs of fixed advertised capacity [42], [15], [8], [7]. It is reasonable to expect many VM types with different types of capacity dynamism will continue to be offered.

**TABLE I**: Representative VM offerings (top) and related work (bottom) shown based on what aspects of our three axes they offer/address. We only show a small subset of the overall related work discussed throughout the paper.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>High</td>
<td>Fine</td>
</tr>
</tbody>
</table>

### Representative VM offerings

<table>
<thead>
<tr>
<th>EC2 on-demand</th>
<th>EC2 spot</th>
<th>EC2 burstable</th>
<th>Google pre-emptible</th>
<th>ProfitBricks</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Classification of related work

<table>
<thead>
<tr>
<th>Price vs. offered capacity</th>
<th>VM- vs. fine-grained scaling</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

- **Granularity of Resource Allocation**: Public cloud providers have generally allowed resource allocation at the relatively “coarse” granularity of an entire VM (or recently, a container) with fixed advertised resource capacities. Whereas the actual resource allocations for these VMs might vary at a finer granularity (recall the aforementioned gap between offered and advertised capacities), it results from actions carried out unilaterally by the provider without involving the tenants (with tenants implicitly inferring such changes through their own performance observations and modeling). A recent trend is the emergence of options for finer scaling starting with very small VMs, e.g., EC2 ElastiCache nodes (1 vCPU, 0.555 GB DRAM). Following this, options for scaling of individual resources for already-procured VMs have started to appear, variously labeled resource-as-a-service [2] and vertical scaling [33]. E.g., Amazon’s burstable instances allow for a VM’s CPU capacity to be scaled up/down dynamically (although via a token bucket like SLA) and its recent C4 instances may alter the P- and C-states of cores. A more explicit option is offered by ProfitBricks wherein additional CPUs may be dynamically added to an existing VM (although dynamically scaling down of CPUs is not offered as far as we know); the tenant is charged based on its CPU x time usage. We show these as offering fine-grained scaling in Table I. It is conceivable that even finer, more flexible, and more diverse (spanning other resources) scaling would be offered.

As Table I (bottom) clarifies, related work has mostly looked at only one and/or two of these dimensions. Novel opportunities for improving the tenant’s cost-efficacy arise by considering these options together. Doing this requires novel algorithmic and system design ideas.
Our Approach: We consider the entire trade-off spectrum identified above in a unified hierarchical framework and study it in the context of a popular/general class of workloads that combine EC2-style spot and on-demand (OD) instances with fine-grained allocation capability. We leverage novel ideas based on data-driven modeling of spot price prediction and fine-grained resource pricing as the building blocks of our framework. We employ myopic control for our optimization objective based on these models that helps overcome the “curses of dimensionality” associated with conventional dynamic programming approaches. Finally, we implement our full control framework as a prototype system based on EC2 and conduct a case study with Memcached with real-world workloads and multi-market spot prices to demonstrate the cost-efficacy of our proposed approaches.

We pick spot instances as our representative of dynamic capacity vs. dynamic offered capacity trade-off for two reasons: (i) they offer real-world dynamic pricing data and infrastructure on which we can validate our techniques, (ii) they represent an extreme in a way for offered capacity dynamism (due to their ON-OFF nature). Our ideas are more generally applicable, however, and adapting them to other points of the overall trade-off space is part of our future work. We pick Memcached as our case study for these reasons: (i) it is an important part of many workloads (e.g., as the caching tier for many persistent data stores), (ii) it is resilient to node failures (although at a performance cost) allowing us to explore the price vs. offered capacity dynamism trade-off, (iii) being open-source and well-documented, it allows us to implement our control algorithms easily and effectively, (iv) it can leverage both scale-up and scale-out (unlike, say, many ACID databases that only benefit from scale-up and may not find offered capacity variations tolerable).

Contributions: Our contributions are:

• Problem Identification: We classify current/emerging VM offerings from commercial cloud providers based on three axis (Table I). As a case study, we pick VM procurement for Memcached using a combination of EC2-style on-demand and spot instances with the additional assumption of fine-grained scaling (which we emulate by running Docker container within a VM and scale the resource allocated to the container at runtime), which covers all dimensions of the grid whereas prior works only focus on one or two axis.

• Modeling: Base on novel insights from analysis using a variety of large-scale spot price traces, we identify the contiguous period of successful bid and the average price during that period as most important and effective quantities to predict for decision making (allowing computationally scalable control design) whereas most related works focus on prediction of actual spot price or long-term statistical properties, e.g, Markovian models that may result in computationally difficult control problems (Section II-A). To incorporate fine-grained resource scaling, we take instance pricing data from Amazon EC2 and Google Compute Engine and employ regression-based estimates for fine-grained pricing model that is used in our framework (Section II-B).

• Framework Design and Integration: We design a hierarchical control framework for tenant’s cost-effective VM procurement. At the top level, a global controller periodically predicts workload properties, updates the aforementioned spot price models and compute the cost-optimal resource allocation and workload partition decisions via an online optimizer. Our optimization formulation scales linearly with the number of spot markets and bids. At the bottom level, we deploy a local controller on each VM which helps deal with unexpected performance degradation in a reactive manner using fine-grained scaling with feedback controllers. Unmet local resource scaling requests are escalated to the global controller for further coordination (Section III).

• Implementation on EC2: We implement our full control hierarchy as a prototype system on EC2. Since currently EC2 does not allow fine-grained scaling, we emulate such effects by running Docker containers within VMs, and scale the resource allocation of containers using “cgroups”. As a case study, we deploy Memcached as a data caching service on our prototype system and implement our workload partition approaches by adapting open-source tools (YCSB [11] and mcrouter [24]) to meet our needs (Section IV).
• **Evaluation:** To demonstrate the cost-benefit of our framework and help understand how the system might work in the real world, we carry out both trace-driven simulation with a variety of long-term spot price trace and real-world workload traces, and live experiments with Memcached modified to run on our EC2-based prototype system. Our findings are promising: (i) In the long-run, our framework can help the tenant save up to 40% costs by partitioning workloads intelligently across spot and on-demand instances facilitated by fine-grained scaling, with high availability (99.92%) of spot instances. (ii) Upon bid failure, our system is able to gradually move data from spot instances to on-demand instances by applying fine-grained scaling with little/minimum performance degradation. (iii) Our reactive control can further help improve application performance under unexpected flash crowds via fine-grained scaling.

II. MODELING METHODOLOGY/ASSUMPTIONS

A. Spot Instances

Plenty of work identifies that spot instances can improve costs. However, their (possibly) lower offered capacity than on-demand instances - itself dependent on the tenant’s bidding policy - may hurt performance. To gainfully use them, a tenant needs to predict well aspects of spot prices relevant to this cost v. performance trade-off.

**Prior Work:** Many papers attempt to predict real-time, near-term spot prices, e.g., via auto-regressive models [1]. Since spot prices could change at a minute’s granularity, such techniques might fail to provide insights on how it might evolve in the long run. A second class of work uses a range of statistical modeling from simple empirical cumulative distributions of key parameters [34, 20] to more complex (with more memory) statistical models [47], perhaps adapting these over time. Although offering an improved treatment of longer term properties than the first class, even if updated adaptively with new observations, they 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 is [32] wherein the “sojourn time” of a given discretized price value is modeled and predicted via a Semi-Markov chain. However, (as we will discuss in Section III) tenant control based on such multi-dimensional models would likely suffer scalability limitations when applied to multiple spot markets with multiple bids for better availability and profitability.

**What Should We Be Modeling?** Rather than model the exact spot price values themselves, we argue that for purposes of capturing cost vs. performance implications, the focus should be on the two features we identify below.

• **Feature I:** Since spot prices tend to be significantly smaller than on-demand prices (of comparably sized instances) during periods when a bid is successful, and since EC2 charges a tenant based on the spot price during such periods (not based on the bid placed by a tenant), attempting to predict spot prices very accurately is of little value. A visual inspection of 90-day long spot price timeseries from four different markets in Figs. 2(a)-(d) and how these prices compare with different bid values (marked in dotted lines) clarifies this. In particular, it suffices that we predict the average spot price during such periods.
with reasonable accuracy (since that is what will determine our costs). To appreciate this, compare two hypothetical predictors P1 and P2 in Fig. 1 that both capture the average equally well but P2 captures the variance better than P1. Since the tenant’s cost estimates (if it were to place the shown bid) are identical with the two predictors, P2 is not necessarily better than P1 despite capturing the exact spot prices better.

- **Feature II:** In fact, P1 is the preferred predictor because it is better at capturing the other important feature, namely, *how long is a successful bid likely to last.* Also, an effective predictor should not over-estimate this quantity - doing so may render a control scheme overly optimistic in its estimation of the associated cost vs. performance trade-off.

![Fig. 2: Four spot price timeseries out of several we use in our evaluation. These were collected during the 90-day period (2015-07-08 to 2015-10-06) and are chosen due to their very different properties.](image)

**Our Prediction Approach:** We analyze a large number of spot price timeseries collected from several markets over several months (only main findings reported here). We find that the key spot price features identified above tend to exhibit a form of short-term temporal locality: over relatively short timescales (a day to a few days, depending on the spot market), they tend to change little, whereas over longer timescales (weeks to months, again depending on the spot market), they undergo more substantial changes. These observations suggest that myopic predictors of these features may perform well. This is a key aspect of spot price modeling missing in existing work.

Our technique employs empirical probability distributions computed over recent sliding time windows \((w\text{ most recent days})\) for making predictions over prediction horizons \((h\text{ upcoming days})\). \(w\) and \(h\) must be chosen such that the aforementioned temporal locality holds. We model as a random variable \(L(b)\) the length of a contiguous period during which the spot price is less than or equal to a bid \(b\). \(L(b)\) captures the lifetime of a spot instance using bid \(b\). We denote as \(p(b) = E[p_t|L(b)]\) a random variable for the average spot price \(p_t\) during a period when the bid \(b\) is successful, \(^1\) which serves to estimate the cost of a spot instance procured by placing a bid \(b\). Figure 1 clarifies these quantities. Large \(L(b)\) and small \(p(b)\) imply long service continuity and low costs, thereby encouraging the use of spot instances under bid \(b\). We use a small percentile (e.g., 5th) of the recently constructed distribution of \(L(b)\) - denoted as \(\hat{L}(b)\) - as our prediction in the ongoing horizon. The reasoning behind this choice is that if the statistical properties of \(L(b)\) do not change much between \(w\) and \(h\), we expect that with a very high probability, bid \(b\) would be successful for at least \((\hat{L}(b))\) time units. We use average of \(\bar{p}(b)\) during relevant \(w\) as its predictor (denoted as \(\hat{\bar{p}}(b)\)) during \(h\).

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

**Validation:** We evaluate our technique with dozens of spot price traces of which we show four 90-day spot price traces for VM types of m3.large and m3.2xlarge in availability zones us-east-1c and

\(^1\)We are overloading the term \(L(b)\) to mean a contiguous duration when bid \(b\) is successful as well as its length. Likewise for the terms \(w\) and \(h\). We are avoiding additional notational complexity since the distinction is very clear based on context.
us-east-1d in Figures 2(a)-(d). We employ \( w=7 \) days and \( h=1 \) day. For each trace, we pick bid price \( b \in \{0.5d, d, 2d, 5d\} \), where \( d \) is the corresponding on-demand price. Table 2 presents our assessments metrics for 16 representative traces. We highlight entries where \( f > 15\% \) or \( \xi > 30\% \) (interpreting these as “poor” predictions). Our evaluation in Section V will reveal that for the purpose of cost-effective tenant procurement, our technique offers good predictive power. Additionally, we will see in Section III, that our modeling greatly simplifies (an otherwise non-trivial) control algorithm design.

<table>
<thead>
<tr>
<th>instance-type</th>
<th>c3.large</th>
<th>r3.large</th>
<th>m3.large</th>
<th>m3.2xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric</td>
<td>( f )</td>
<td>( \xi )</td>
<td>( f )</td>
<td>( \xi )</td>
</tr>
<tr>
<td>market 1: us-east-1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0.5d )</td>
<td>0.0947</td>
<td>0.0703</td>
<td>0.0756</td>
<td>0.2900</td>
</tr>
<tr>
<td>( d )</td>
<td>0.0698</td>
<td>0.0698</td>
<td>0.0927</td>
<td>0.2392</td>
</tr>
<tr>
<td>( 2d )</td>
<td>0.0690</td>
<td>0.0701</td>
<td>0.0802</td>
<td>0.4256</td>
</tr>
<tr>
<td>( 5d )</td>
<td>0.0698</td>
<td>0.0714</td>
<td>0.0714</td>
<td>0.3466</td>
</tr>
<tr>
<td>market 2: us-east-1c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0.5d )</td>
<td>0.1014</td>
<td>0.0586</td>
<td>0.0519</td>
<td>0.3460</td>
</tr>
<tr>
<td>( d )</td>
<td>0.1238</td>
<td>0.1120</td>
<td>0.0509</td>
<td>0.3567</td>
</tr>
<tr>
<td>( 2d )</td>
<td>0.1456</td>
<td>0.1097</td>
<td>0.0858</td>
<td>0.7027</td>
</tr>
<tr>
<td>( 5d )</td>
<td>0.1373</td>
<td>0.1272</td>
<td>0.0728</td>
<td>1.1451</td>
</tr>
<tr>
<td>market 3: us-east-1d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0.5d )</td>
<td>0.0920</td>
<td>0.0553</td>
<td>0.0696</td>
<td>0.0142</td>
</tr>
<tr>
<td>( d )</td>
<td>0.0482</td>
<td>0.0725</td>
<td>0.0861</td>
<td>0.1990</td>
</tr>
<tr>
<td>( 2d )</td>
<td>0.0482</td>
<td>0.0725</td>
<td>0.0968</td>
<td>0.5204</td>
</tr>
<tr>
<td>( 5d )</td>
<td>0.0482</td>
<td>0.0729</td>
<td>0.0897</td>
<td>0.8901</td>
</tr>
<tr>
<td>market 4: us-east-1e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0.5d )</td>
<td>0.0244</td>
<td>0.0574</td>
<td>0.0857</td>
<td>0.1441</td>
</tr>
<tr>
<td>( d )</td>
<td>0.0241</td>
<td>0.0554</td>
<td>0.0673</td>
<td>0.1565</td>
</tr>
<tr>
<td>( 2d )</td>
<td>0.0241</td>
<td>0.0554</td>
<td>0.0642</td>
<td>0.1577</td>
</tr>
<tr>
<td>( 5d )</td>
<td>0.0241</td>
<td>0.0562</td>
<td>0.0741</td>
<td>0.1609</td>
</tr>
</tbody>
</table>

TABLE II: Evaluation of our prediction technique for four different markets and four different advertised instance sizes.

B. Fine-Grained Scaling

Offering explicit fine-grained scaling knobs to tenants may bring several efficiency benefits to the cloud provider and various aspects of this have been studied recently [2], [33]. Getting explicit hints about tenants’ near-term needs may allow the provider to better consolidate tenant workloads. It may also allow the cloud provider to monetize otherwise “fragmented” resources.

Correspondingly, tenants may find that such knobs allow them to avoid procuring excessive resources which, depending on the pricing scheme for such resource procurement, may save them costs compared to procuring resources only at the granularity of entire VMs. As a motivating example, Figure 3 shows a simplified scenario by scaling the dynamic arrival rates and working set sizes of the wikipedia access trace [35] (Figure 8) with 5GB, 100GB, 200GB dataset, respectively, and turning into resource demands of CPU, RAM, and network bandwidth. Then we solve an ILP which minimizes the VM costs while satisfying resource demands, and obtain the cost-optimal combination of VMs based on all Amazon EC2
VM types. The resource wastage, compared with “just-enough” resource to serve the workload, is shown in Figure 3. We observe memory and CPU wastage under 5GB dataset, CPU and network bandwidth wastage under 200GB dataset, but negligible wastage under workload with 100GB dataset. For such a tenant, the key trade-offs to navigate would be: (i) how to exploit fine-grained scaling to avoid resource wastage while still meeting its own performance target, (ii) how to leverage workload predictability and react to sudden changes promptly and (iii) how to coordinate fine-grained vs. coarse-grained scaling (fine-grained scaling cannot be done infinitely due to application scalability limitations and physical resource capacity).

![Resource wastage graph](image)

Fig. 3: Resource wastage by using a cost-optimal combination of EC2 instances (derived from an ILP that minimizes the VM costs while satisfying total resource demand) to serve a data caching workload scaled from Wikipedia access trace [35], compared with provisioning “just-enough” resources to serve the workload. The working set size is scaled to 5GB, 100GB and 200GB to create workloads with different resource demand ratios.

Although collectively today’s providers likely already offer the full spectrum of VM procurement choices shown in Table I, no single cloud provider does so all by itself. Consequently, for incorporating this aspect into our decision-making, we must assume a reasonable API for such scaling and a corresponding pricing structure.

- **API**: We assume that our cloud provider’s API contains facilities for fine-grained CPU (at the granularity of 1 vCPU), memory, and network IO scaling of an existing VM’s capacity.
- **Pricing**: An important concern is the assumption to make about how individual fine-grained resources might be priced by such a cloud provider. Generally speaking, we expect that larger allocation requests for a given resource within a VM might be less expensive per-unit capacity analogous to the buy-in-bulk effect exhibited by many commodities. Additionally, in such environments prices might also reflect the current/near-term expected resource competition on servers (e.g., higher on busier servers). We leave more rigorous modeling and design of fine-grained resource pricing for future work and take a simpler approach guided by our observations of current VM prices. As a representative example, we choose sample VM prices corresponding to different VM types (with different CPU and memory capacities) offered by Google Compute Engine [17], and fit VM price as a function of number of vCPUs and amount of RAM via linear regression as shown in Figure 4(a). We find that the resulting plane $p^G = 0.0246 \cdot c + 0.0036 \cdot m$ fits the VM prices very well with $R^2 = 0.9992$.

Similarly, we take the same approach and obtain $p^G = 0.0412 \cdot c + 0.0061 \cdot m$ for Amazon EC2 with $R^2 = 0.9992$ (Figure 4(b)).

### III. Tenant Resource Procurement

We design a two-timescale hierarchical control framework shown in Figure 5 for our tenant VM procurement. At a coarser time-scale (e.g., hours), the online optimizer making up our “predictive control”

---

2We are admittedly being somewhat non-rigorous here: a key assumption of linear regression that the independent variables (number of vCPUs and RAM capacity of a VM) be independent of each other does not quite hold - generally, larger (smaller) VMs have larger (smaller) amounts of both resources.
exploits predictability within the workload (e.g., arrival rates, working set size) and spot price features \((L(b)\) and \(\bar{p}(b)\)) to determine: (i) the type and amount of additional resources 3 to procure (or reallocate) and (ii) partitioning of its “state” among its on-demand vs. spot instances. The global controller translates the output of the predictive control into actual resource procurement decisions combining VM-level and fine-grained options. The global actuator exercises appropriate knobs in the cloud API to issue requests to the cloud. Generally speaking, additional actions inside the tenant application might need to be taken by the actuator to complement these resource procurement actions. E.g., after scaling up the number of vCPUs, the actuator might increase the number of service threads of a Web application to fully utilize the newly added CPU capacity. We do not explore this last aspect in our design and implementation leaving it to future work.

At a finer time-scale (e.g., every 10 min), each existing VM collects performance statistics and makes local control decisions to continue meeting performance targets in the face of prediction errors, e.g., unpredictable flash crowds, under/over-provisioning of VMs by predictive control, system noise (e.g., VMs do not get enough capacity due to cloud’s resource overbooking or statistical multiplexing). This constitutes our “reactive/corrective control”. The actual resource scaling requests are issued to the cloud by a local actuator. If some local resource scaling requests cannot be fulfilled by the cloud (e.g., due to lack of capacity on the physical host), or if there is not enough “headroom” for local scaling-up 4, an escalation will be triggered wherein the global coordinator will invoke the global controller to take remedial actions using its global view of resource assignments and individual VM’s performance.

When bids fail, the global coordinator is responsible for taking remedial action. One set of options, could be based on checkpointing spot instances as explored in some recent papers (reactively upon receiving a bid failure warning if this much time suffices for the concerned state or proactively/incrementally) [46], [38], [34]. Given the nature of the workload we focus on, we choose the option of re-directing the workload served by affected spot instances to existing on-demand instances and scaling up resource capacity of on-demand. Again, an escalation is needed when there is not enough headroom and new VMs are needed, however, with delay of VM boot time that is comparable to or even higher than the failure warning period [27].

In what follows, we describe only selected aspects of our overall system due to space limits. In particular, the idea of combining complementary predictive and reactive elements into a hierarchical control framework is a very well-established design principle numerous specific forms of which have been studied [37], [29], [31], [16]. We conclude the section with a discussion of some key implications of our design and some future directions.

3We consider CPU and memory in our formulation and will generalize it to include other resources in future work.

4We define the headroom as the difference between the scalability limits of an application and the amount of existing resource on a single VM. E.g., if an application does not scale up beyond four cores, and there are already three cores in the VM, the headroom is one core.
A. Predictive Control

Real-world workloads exhibit varying degrees of predictability due to phenomena such as time-of-day, day-of-week, and other seasonal effects. One can usually extract predictable components, often of significant relative magnitude, via one of several techniques that have been studied. Conventional approaches for online optimal control (e.g., Markov Decision Processes) are based on devising stochastic dynamic programs (DP) that exploit such predictability in problem inputs. However, the vanilla DP-based approach is unsuitable for our tenant procurement due to scalability limitations posed by the well-known “curses of dimensionality.” First, although workload properties can likely be captured via low-order predictors (as is the case for the workloads in our evaluation), related work does not offer a clear verdict on spot price prediction and some evidence suggests that this likely requires complex modeling [32]. Even if it were possible to predict these well, one would run into a big computational hurdle due to the large number of spot price markets, evolving in many different ways (recall examples in Fig. 2). To fully exploit the complementary price vs. performance trade-offs offered by these markets, a DP-based approaches would need to maintain a “state” whose dimension could be \( O(N_{\text{markets}} \times N_{\text{vmtypes}} \times |B|) \), where \( N_{\text{markets}} \), \( N_{\text{vmtypes}} \), and \( |B| \) represent the number of markets, number of instance types, and cardinality of the set of bids the tenant uses. Clearly, we need a way to simplify this.

**Key Idea A:** We employ a “myopic” control based only on short-term forecasts wherein we use a notion of state that simply remembers, for each existing instance, its “residual life” (wherein its lifetime at procurement time is taken to be \( \hat{L}(b) \), \( b \) being the bid used to procure it, is easily updated at each control window of size \( \Delta \) if the instance is still alive). Furthermore, to model the negative implications (performance-wise or explicit cost-wise) of the inevitable failure of a spot instance in a finite amount of time, we devise the notion of a “loss rate” (costs divided by \( \hat{L}(b) \), to be described below) that we assume is incurred uniformly over the instance’s lifetime. On-demand instances have \( \hat{L}(OD) = \infty \), implying 0 loss rate.

1) **Our Predictive Optimizer:** We consider a slotted time system with time slots of length \( \Delta \) (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” (in-memory content/state) size \( \hat{M}_t \) during the next time slot, procures VMs from multiple EC2 markets with appropriate bids, and devises a complementary partitioning of the working set. The goal of our formulation is to minimize the tenant’s

---

**Fig. 5:** Overview of our hierarchical control framework. To keep the figure clear, we the control flow during bid failures.
costs while satisfying performance target during the next time slot.

**Workload Partitioning:** Workload placement techniques are designed to capitalize on knowledge about the popularity distribution of content which tends to be heavy-tailed with a small portion of the working set being popular (“hot”) and a large portion being unpopular (“cold”). Conventional wisdom would suggest that we employ the following intuitively appealing approach: ensure that the hot data resides on the (highly available) on-demand instances and the cold data on spot instances (that may fail). While certainly offering cost savings over an approach based only on using on-demand instances, this approach misses out on additional important cost savings opportunities: it would tend to leave memory capacity in on-demand instances under-used (since hot content is small in size but is accessed by many requests resulting in low memory but high CPU needs); similarly, it would tend to leave CPU resources on spot instances under-used. Whereas fine-grained de-allocation knobs may help alleviate this to some extent, their efficacy might be limited by both the granularity of modulation allowed as well as the cloud provider’s ability at a given time to offer such scaling (up or down). Rather than relying solely on fine-grained scaling, we would like our placement technique itself to address this resource usage imbalance, and offer corresponding savings.

**Key Idea B:** We devise a workload partitioning 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 6 provides an illustration of the basic idea using a hypothetical heavy-tailed content popularity distribution. We formalize this idea next.

Denote as $B$ the set of all bid values. Denote as the set of all markets $S = \{\text{regions}\} \times \{\text{availability zones}\} \times \{\text{VM types}\} \cup \{\text{On-demand}\}$. 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 ($b = OD$) and the lifetime $L(OD) = \infty$.

Denote as $x^s_t, y^s_t \in [0, 1]$ the portions of hot and cold data that will be placed in market $s \in S$ using a bid $b \in B$, respectively. Denote as $R = \{vCPU, RAM\}$ the set of basic resources. Denote as $N^r_{tsb}$ and $\tilde{N}^r_{tsb}$ the amounts of existing resource (“system state”) and additional resources to be procured (“control actions”) at the start of time slot $t$, respectively, for resource type $r \in R$ in market $s \in S$ with bid $b \in B$. $N^r_{tsb}$ could be negative (corresponding to a need for de-allocation). Denote as $H$ the portion of working set that is hot $^5$ and $\hat{M}_t$ the predicted working set size in time slot $t$. Assuming the whole predicted working set needs to be held in memory (a reasonable choice for memcached, our case study in Section V; perhaps wasteful in memory for some other workloads), we have

$$\sum_{s \in S} \sum_{b \in B} x^s_t = H, \quad \forall s \in S, b \in B$$

$$\sum_{s \in S} \sum_{b \in B} y^s_t = 1 - H, \quad \forall s \in S, b \in B$$

$$N^r_{tsb} + \tilde{N}^r_{tsb} = (x^s_t + y^s_t)\hat{M}_t, \quad r = \text{RAM}, \quad \forall s \in S, b \in B$$

$$x^s_t + y^s_t \geq \zeta, \quad s = \text{on-demand}$$

where the last constraint guarantees that at least $\zeta$ fraction of the entire working set would be placed on on-demand instances for which scale up can be attempted (faster than starting a new instance) immediately upon one or more bid failures. $H$ is application/workload-specific and could also be time-varying.

---

$^5$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.
Define $l_t = \phi(\lambda_t, \text{vCPU}, \text{RAM})$ to capture how the resource allocation affects application performance wherein $\lambda_t$ and $l_t$ are the workload arrival rate and a relevant performance metrics (e.g., latency for our memcached case study), respectively. The $\phi(.)$ function can be either empirically measured offline (as is done in our evaluation in Section V) and updated periodically online (i.e., $\phi(.)$ would either be a regression model or a lookup table based on the performance profiling results), or theoretically modeled, e.g., via queueing analysis [36], [16]. Define $F(.) \in [0, 1]$ as the CDF of data popularity distribution (measured and updated by the tenant in an online manner). We have
\[ \phi(\lambda_t(x_t^s F(H) + y_t^s (1 - F(H))), (\hat{N}^csb_t + \hat{N}^{smb}_t) (N^{msb}_t + \hat{N}^{msb}_t)) \leq l^{TGT}, \]
wherein $c = \text{vCPU}$ and $m = \text{RAM}$, $N^{csb}_t$ and $N^{csb}_t$ are integer variables.

**Resource Costs:** Denote as $\hat{p}_t^{rsb}$ the predicted average price for resource $r$ during $t$. Therefore, the total resource costs (with a slight over-estimation due to ignoring bid failures during this time slot) can be expressed as $\sum_{s \in S} \sum_{b \in B} \sum_{r \in R} \Delta \hat{p}_t^{rsb} (N^{rsb}_t + \hat{N}^{rsb}_t)$.

**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. As mentioned earlier, 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 VM 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 build upon the idea of (barring hardware caching/TLB-related effects), if fine-grained scaling is enabled. However, memory, resource that can be scaled up/down instantaneously without affecting application performance much, may need to be predicted via appropriate predictive models, e.g., an AR(2) model $\hat{\lambda}_t = \alpha_1 \lambda_{t-1} + \alpha_2 \lambda_{t-2}$, before solving the optimization problem.

**Optimization Problem Formulation:** At the beginning of time slot $t$ (proactive control window), we solve the following online control problem:
\[
\text{Minimize}_{x_t^s, y_t^s} \sum_{s \in S} \sum_{b \in B} \left( \sum_{r \in R} \Delta \hat{p}_t^{rsb} (N^{rsb}_t + \hat{N}^{rsb}_t) \right)
+ \eta \max \{0, -\hat{N}^{msb}_t \}
+ \Delta (\alpha x_t^s + \beta y_t^s) \hat{M}_t \\
\frac{\hat{L}^s(b)}{L^s(b)}
\]
s.t. (1), (2).

Note that $\hat{\lambda}_t$ and $\hat{M}_t$ need to be predicted via appropriate predictive models, e.g., an AR(2) model $\hat{\lambda}_t = \alpha_1 \lambda_{t-1} + \alpha_2 \lambda_{t-2}$, before solving the optimization problem.

**B. Reactive Control Design**

To deal with the less predictable component of the workload as well as various sources of system noise within a coarse-grained time slot used in the above optimization formulation, the tenant could carry out...
fine-grained resource scaling at a finer time granularity (e.g., one minute). We denote as $t'$ the control window of fine-grained resource scaling.

Our reactive control is done in a decentralized fashion, i.e., we deploy one local reactive controller within each instance. This reactive controller collects performance-related statistics at both application level (e.g., application latency, miss rate) and system level (e.g., CPU utilization, used RAM capacity) and makes local resource scaling decisions to continue meeting performance targets in the face of prediction errors, e.g., unpredictable flash crowds, under/over-provisioning of resources by predictive control, system noise (e.g., VMs do not get enough capacity due to cloud’s resource overbooking or statistical multiplexing). The local decisions are then passed on to the cloud via the global controller; any unmet resource scaling requests (e.g., due to lack of physical resource capacity) will also be coordinated by the global controller, in which case extra VMs might be needed. Next, we describe our detailed feedback control based fine-grained scaling. We denote as $m_{t'}$ and $c_{t'}$ the amount of RAM and number of vCPUs at the start of $t'$ for local reactive control; the superscripts for market $s$ and bid $b$ are omitted for simplicity.

**Memory scaling.** We periodically monitor the latency $l_{t'}$ of a memcached server. It may exceed the target latency $l^\text{TGT}$ when the server is offered less RAM capacity than it actually needs to hold all the working set in memory (possibly due to mis-prediction) and the miss rate worsens. We introduce latency feedback as a correction for mis-prediction as follows (assuming that the prediction error changes smoothly):

$$m_{t'} = m_{t'-1}(\zeta^m l_{t'-1} - l^\text{TGT} + 1)$$

where $\frac{l_{t'-1} - l^\text{TGT}}{l^\text{TGT}}$ reflects the relative latency violation. $w^m(>0)$ is a scaling factor used to mitigate the oscillations incurred by feedback control. If measured latency exceeds the target in time slot $t' - 1$, the controller will further scale up RAM capacity in time-slot $t'$ according to the relative latency violation, and vice versa.

However, sometimes adding performance feedback may not suffice due to the “stateful” property of memory. This problem arises when keys evicted from memory in previous time slots are accessed again in future time slots when scaling down, which requires a second read request to the slow persistent back-end database and an insert operation to add the key-value pair into Memcached, thereby causing performance degradation. Toward this, we “pro-actively” add dynamic slackness as follows (especially when scaling down):

$$m_{t'} = m_{t'-1}(w^m l_{t'-1} - l^\text{TGT} + 1)(1 + s_{t'})$$

wherein the dynamic slackness $s_{t'} (0 \leq s_{t'} < 1)$ is determined based on the following heuristics: If the key popularity does not change much across time slots, there is a higher chance that keys accessed in previous time slots might be accessed again in future time slots. Therefore, the tenant should be more conservative when scaling down RAM capacity. To capture the key popularity change, we further define “turnover” $O_{t'}$ as the number of new keys plus the number of evictions during time slot $t'$. When scaling down RAM capacity, i.e., $\hat{M}_{t'} < M_{t'-1}$, if the relative turnover $\frac{O_{t'} - 1}{\hat{M}_{t'-1}}$ is smaller than a pre-specified threshold (which implies that the key popularity might not change much) we add more slackness. Otherwise, we add less (or no) slackness. In our evaluation, we expect to see that dynamic slackness together with performance feedback can be used to compensate for the impacts of mis-prediction, multi-tenancy and other random/unknown system noise that might affect application performance.

**CPU scaling.** Similar to memory scaling, we add latency feedback to CPU scaling to compensate for mis-prediction and system noise:

$$c_{t'} = [c_{t'-1}(w^c l_{t'-1} - l^\text{TGT} + 1)]$$

Note that in our current design, CPU resource is controlled at the granularity of cores (discrete variable)
whereas memory capacity is considered as a fluid variable. Therefore, high oscillations might occur in both the control decisions and performance measurements. To avoid such undesirable behavior, we make the scaling factor $w'_c$ dynamic: When $l'_t << l^{TGT}$ that results in scaling down decisions, we set $w'_c$ to a smaller value (being conservative) whereas $w'_c$ can be larger when $l'_t > l^{TGT}$.

Note that due to the “stateless” property of CPU resource, there is no need to add dynamic slackness when scaling down.

Combining memory and CPU scaling. For each memcached server, we monitor the average per-core CPU utilization and miss rate in each time slot, which are commonly considered as indicators of CPU and memory resources being scarce or abundant. We set lower/upper bounds for the two metrics separately. Then we identify the bottleneck resource(s) based on the following heuristics: we add performance feedback to CPU scaling controller if (i) the average per-core CPU utilization exceeds the upper bound which indicates that CPU resource might have become the bottleneck or (ii) it falls below the lower bound, implying possible surplus of CPU resource. Otherwise, the CPU utilization falls between the two bounds, we don’t carry out fine-grained CPU scaling for this memcached server. Similar heuristics can be applied to memory scaling controller by monitoring the miss rate. When both CPU and memory are bottlenecks, attributing performance degradation among bottleneck resources itself become a non-trivial problem. In evaluation, we choose to add latency feedback equally to both resources and leave more complex scenarios to our future work.

C. Salient Features and Alternatives

Two key implications of our design choice are worth discussing. First, it would appear that due to the pricing scheme we assume (based on validation from multiple providers - ProfitBricks, Google, and Amazon - as described in Section II-B), our control technique would always attempt to first exercise fine-grained allocation knobs (on existing VMs) before resorting to VM-level scaling. While this is largely true, one reason why our control might depart from this is due to a better availability vs. price trade-off (captured by the bid failure penalty term) offered by procuring a new VM than adding resources to an existing one. It is possible that more complex pricing (e.g., incorporating volume/buy-in-bulk discounts as seen for other commodities) may emerge in which case the choice between fine-grained vs. VM-level scaling would become more complex. Our future work will consider these issues.

Second, while our modeling ideas of an instance’s residual lifetime and a rate describing loss incurred uniformly over its lifetime allow us to overcome scalability limits in the design of our control, this simplification has some downsides. In particular, during a given instance of decision-making, our predictive control would likely pick spot instances only from one market and based on a single bid. This can create pools of spot instances that will “fail” together. In our evaluation, we observe that over time different markets are chosen (due to dynamically evolving spot price features), thereby helping alleviate some of these failure correlations. We plan to explore additional enhancements that deliberately try to spread spot instances across multiple markets/bids in ways that are informed by data-driven correlation modeling. Similar correlation studies in recent work can offer us useful insights towards this [45], [34], [18], [20].

IV. IMPLEMENTATION

We describe the different parts of our framework using a prototype implementation based on Memcached. As shown in Figure 7, we use Yahoo Cloud Serving Benchmark (YCSB) based clients to generate requests to back-end Memcached servers. Facebook’s mcrouters [24], Memcached protocol routers, are used in front of our servers to partition the data and rebalance the load across the servers. The framework consists of the following components:

Key Partitioner: We create bloom filters using access frequency-based heuristics which are refreshed after regular time periods to keep track of “hot” keys (access frequency above a certain threshold within a given time window). More sophisticated heuristics such as ghost lists [25], probabilistic counters [14], Count-Min Sketch [12], etc., can also be used. This information is used to annotate keys generated at
YCSB clients as hot (append a prefix “h”) or cold (append prefix “c”). Similar annotation can be done by embedding an annotator at mcrouter layer as well. 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 physical server separation.

**Load Balancer:** Periodically, mcrouter updates its records of the hot and cold weights of all Memcached instances based on the output of optimization problem solved by **Global Controller** (described below). Within each virtual pool, it uses weighted consistent hashing algorithm to forward the requests such that the amount of hot/cold data placed on the Memcached servers are proportional to their hot/cold weights.

**Local Reactive Controller:** We add a **Local Reactive Controller** to each Memcached server that periodically checks its own resource utilization and statistics about miss rate, arrival rate, used memory capacity, total items, etc. This local controller can also submit fine-grained resource scaling requests to the cloud via the **Global Controller** to handle unexpected workload variations reactively. Since Amazon EC2 does not support fine-grained resource scaling without restarting a VM, we create a Docker container within each VM and run Memcached as a service inside this container. Thus, the local controller running in the VM is responsible for scaling the resource allocations of the container (using “cgroups”) based on its local reactive decisions or commands from the **Global Controller**.

**Global Controller:** The **Global Controller**, running as a separate VM in the cloud, periodically solves the online optimization problem (Section III-A) and determines the hot $x_i^b$ and cold $y_i^b$ weights for the market $s$ with bid $b$. This information is then sent to all the load balancers (mcrouters). 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. If fine-grained resource scaling is enabled and instances do not have the same resource capacity, we re-allocate weights proportional to their DRAM capacities. For a multi-tier application, the **Global Controller** is also responsible for coordinating the resource scaling requests across different tiers, e.g., scaling up the CPU capacity of the database tier vs. scaling up the RAM capacity of the caching tier.

Furthermore, in case of bid failures, it triggers reactive control mechanism. Within the two-minute bid failure warning period, it gradually increases the weights of on-demand instances while scaling up their resource capacities as well. This may also require starting extra on-demand instances if fine-grained scaling

---

6In this work, we assume a reliable mechanism to commit this information consistently across all mcrouters. Systems such as Chubby [3], Zookeeper [19] can be used for this purpose.
fails (maximum scaling limits of the VM host) or due to application specific reasons (e.g., Memcached’s scale-up issues [4]).

V. Evaluation Results

In this section, we take Memcached as a case study to demonstrate the effectiveness of our proposed control framework. First, to demonstrate the cost-efficacy of the proposed workload partition scheme (online optimization) in the long run, we conduct trace-driven simulation by scaling a real-world workload trace and three-month spot price traces from Amazon EC2. Second, we implement the full control framework as a prototype system (cf. Section IV) on Amazon EC2. We run Memcached on our prototype system with 24-hour spot price traces to demonstrate the short-term performance, in particular, how the system reacts to unexpected bid failure and the corresponding implication on performance. Third, we add our distributed reactive control into the system and show scenario when the reactive control helps improve application performance via our feedback control based fine-grained scaling.

Fig. 8: (a) (b) Normalized working set size and request arrival rates from Wikipedia access trace [35]. The R-Squared of the AR(2) prediction is 0.99 and 0.94 for (a) and (b), respectively. We scale them to create our synthetic workloads.

A. Experiment setup

**Spot market.** We assume that the tenant uses m3.large as spot instance type across availability zones of us-east-1c and us-east-1d (denoted as s1 and s2) with bids of $b = d, 5d$ (denoted as b1, b2 respectively) in each zone where $d$ is the on-demand price. Therefore, we can mark four markets: $s1b1, s1b2, s2b1$ and $s2b2$. The spot price traces are shown in Figure 2(a)(b). We allow fine-grained scaling on on-demand instances but not spot instances.

**Workload.** We generate our workload based on the dynamic arrival rates $\lambda_t$ and working set size $M_t$ from Wikipedia access trace [35]. We find that both $\lambda_t$ and $M_t$ can be well captured via prediction techniques, e.g., AR(2) model where $\lambda_t = \gamma_1 \lambda_{t-1} + \gamma_2 \lambda_{t-2}$ and similar predictor for $M_t$. We scale the wikipedia access trace in Figure 8 such that the peak arrival rate becomes 325kops, which requires 20 vCPUs according to our offline profiling experiments in order to meet the latency target, and the maximum workingset size becomes 60GB. We set $\zeta = 0.05$ such that at least 5% of the overall working set will be placed on on-demand instances at any time.

**Baselines.** Denote as “prop” our proposed workload partition approach. We further create two baselines to compare against: (i) “BL-1”: all data are stored on on-demand instances (no bidding). (ii) “BL-2”: we store all hot data on on-demand instances and leave all the cold data to spot instances. In all baselines, the workload partition (if needed) on spot instances across markets under different bids is determined by solving the online optimization problem in Section III.

B. Trace-driven simulation

Since we only have 24-hour workload traces, we repeat the trace and add 20% uniformly distributed random noise to both the arrival and workingset size to create a 90-day workload trace. We conduct experiments with a variety of spot price traces and only show a subset of results here due to space limit. We show the total costs break-down and CPU-Hour break-down under different baselines in Figure 9.
We have several observations. First, we find that Prop is able to save as much as 40% costs compared to BL-1 where only on-demand VMs are used. Prop also outperforms other baselines by 20% to 25% in total costs. This is because BL-2 always place all hot data on on-demand instances, even when the predicted $L(b)$ (the contiguous period during which bid $b$ is successful) is long enough such that the cost-benefit of using spot instance outweighs the possible “loss” (cf. Section III for the loss rate function), whereas Prop is able to leverage such information and procure spot instances to shed costs. Second, we observe that Prop uses 9.5% more spot resources and total resources than baselines (possibly also yield better performance), which is due to the fact that spot price can be as low as 10% of the on-demand price and the tenant can still save costs even if they use more resource. Third, we see that (though results not shown here) when the spot price fluctuates quite often between low and high values, the tenant would only use on-demand instances. This (to some extent) demonstrates the efficacy of our spot price model for $L(b)$: Since we only use the 5%ile out of a history window as prediction of $L(b)$, the predicted value would be very low in such scenarios (implying near-term bid failure), which prohibits the tenant from using spot instances. The overall service availability probability is 99.92% across three months.

![Figure 9](image_url)  
Fig. 9: (a) Total costs (norm. against BL-1) and (b) normalized CPU·Hour (norm. against on-demand CPU·Hour) break-down under different strategies.

**Key insights.** (i) Prop achieves high cost-savings (40%) by partition workload intelligently among on-demand and spot instances. (ii) Prop could provide better performance by procuring more spot resources when predicted $L(b)$ is large enough such that the cost-saving outweighs bid failure loss. (iii) Our spot price model with $L(b)$ can effectively prohibit usage of spot instances when spot price exhibits high variations, and provides 99.92% availability in our evaluation.

### C. Benchmark on EC2: Cost-efficacy of workload partition

To demonstrate how our proposed framework work in the real world and its impact on application performance, we conduct experiments with Memcached (data caching service) as a case study on our prototype system on EC2 (cf. Section IV) using the 24-hour workload trace shown in Figure 8, and a 24-hour spot price trace taken from m3.large (Figure 10). We use YCSB to generate workloads with key popularity following a Zipfian distribution (Zipfian constant equal to 0.99). By default, the workload has 100% READ operations and Memcached is initialized as an empty cache. Upon a READ miss, the client has to look up the key in the remote DB and insert the key-value pair into Memcached. Note that the workload is scaled such that network bandwidth does not become the performance bottleneck.

We observe that in this 24-hour period, no instances are procured from availability zone us-east-1c with bids of $d$ or $5d$, possibly due to smaller $\hat{L}(b)$.

We show the application performance (24-hour average latency) under different strategies in Figure 11(a). We find that although Prop is able to offer comparable average and 95%-le to our baselines. Fig. 11(b) shows the latency degradation due to a bid failure. We find that our reactive control is able to overcome the effect of the reduced capacity by re-directing the “lost” workload to existing on-demand instances and increasing their CPU and RAM capacity using fine-grained scaling.
Fig. 10: 24-hour spot price of m3.large from two markets with bids equal to on-demand and five times on-demand price. A bid failure occurs during the 15-th hour at the market of s2b1 (availability zone us-east-1d with bid equal to on-demand price).

Fig. 11: (a) The average latency under different strategies. "*" reflects the 95%ile of latency values in each hour. (b) Latency during/after bid failure.

D. Benchmark on EC2: Reactive control with fine-grained scaling

Now let us focus on the performance after adding reactive control with fine-grained scaling. We make the arrival rate in the 24-th hour three times as high as the original arrival rate in order to mimic the effect of an unexpected flash crowd which results in latency violation that cannot be handled by predictive control. To see how fine-grained scaling helps in reactive control, we create a baseline strategy ElastiCache which can only use the VM type of cache.t2.small from EC2 (1 vCPU and 1.55GB RAM) for reactive control but not fine-grained scaling. We expect to see that reactive control is able to handle unexpected flash crowd and fine-grained scaling further helps improve performance more swiftly than coarse-grained scaling.

We show the application performance under ElastiCache and Prop in Figure 12. We observe that after doing the predictive control, both strategies experience slightly higher latency due to load re-distribution among servers and cold cache warm-up (first get request to a new key will result in a get-miss and a second get request to the same key is served by the remote DB). Then the latency violation occurs after the flash crowd suddenly comes. At around 680-th second, both strategies detect the performance degradation and start to do reactive control. Prop scales up CPU resource on existing VMs almost immediately and the latency drops below latency target very quickly, whereas ElastiCache has to wait for around 100 seconds (VM boot delay) for new cache nodes to become available. The newly started VM has to be warmed-up whereas no cold cache warm-up is needed in Prop since we only need extra CPU resource and there is

Fig. 12: Performance comparison of ElastiCache vs. Prop with reactive control and fine-grained scaling during the flash crowd period.
enough headroom in the existing VMs. However, when there is not enough CPU capacity headroom in the existing VMs or the existing ones have reached the application’s scalability limits (meaning adding more resource capacity to an instance does not improve performance), new VMs still need to be started.

**Key insights:** (i) Reactive control helps handle the sudden burst of the workload. (ii) With enough headroom in existing VMs, fine-grained resource scaling takes effect more responsively/faster than coarse-grained VM scaling (and no need for cache warm-up as seen in the baseline), thereby resulting in better performance. (iii) However, when the headroom in existing VMs is not enough, even fine-grained resource scaling does not suffice, and new VMs have to be started which incurs performance degradation due to VM boot delay and cold cache warm-up.

**VI. Conclusion**

We identified three key qualitative axes using which these trade-offs among numerous VMs offered by public clouds can be succinctly expressed: price dynamism, offered capacity dynamism, and granularity of resource scaling. We argued that related work lacked techniques for effectively navigating this trade-off space. To address this gap, we devised modeling and control techniques that combined Amazon EC2 style spot and on-demand instances with fine-grained resource allocation knobs that have begun being offered by public clouds. We implemented an Memcached prototype tenant that procured VMs from Amazon EC2 using our techniques. Using extensive evaluation based on real-world workload and spot price traces, we demonstrated the efficacy of our approach in reducing costs while meeting performance goals.
REFERENCES


