Dual Scaling VMs and Queries: Cost-effective Latency Curtailment

Juan F. Pérez*, Robert Birke†, Mathias Björkqvist† and Lydia Y. Chen†
*Universidad del Rosario, Colombia. juanferna.perez@urosario.edu.co
†IBM Research Zurich, Rüschlikon, Switzerland. Email: {bir,mbj,yic}@zurich.ibm.com

Abstract*—Wimpy virtual instances equipped with small numbers of cores and RAM are popular public and private cloud offerings because of their low cost for hosting applications. The challenge is how to run latency-sensitive applications using such instances, which trade off performance for cost. In this study, we analytically and experimentally show that simultaneously scaling resources at coarse granularity and workloads, i.e., submitting multiple query clones to different servers, at fine granularity can overcome the performance disadvantages of wimpy VM instances and achieve stringent latency targets that are even lower than the average execution times of wimpy servers. To such an end, we first derive a closed-form analysis for the latency under any given VM provisioning and query replication level, considering cloning policies that can (not) terminate outstanding clones with (without) an overhead. Validated on trace-driven simulations, our analysis is able to accurately predict the latency and efficiently search for the optimal number of VMs and clones. Secondly, we develop a dual elastic scaler, DuoScale, that dynamically scales VMs and clones according to the workload dynamics so as to achieve the target latency in a cost-effective manner. The effectiveness of DuoScale lies on the observation that the application performance only scales sub-linearly with increasing vertical or horizontal resource provisioning, i.e., resources per VM or number of VMs. We evaluate DuoScale against VM-only scaling strategies via extensive trace-driven simulations as well as experimental results on a cloud test-bed. Our results show that DuoScale is able to achieve the stringent target latency by using clones on wimpy VMs with cost savings up to 50%, compared to scaling brawny VMs that have better performance at a higher unit cost.

I. Introduction

Resource elasticity is a central operation in the cloud computing paradigm, as scaling the number of provisioned VMs ensures application performance in a cost-effective manner under all load conditions. The cost of renting VM instances in the cloud becomes ever lower, in many cases at the expense of conditional capacity guarantees, e.g., the Amazon micro instances [1], or resource oversubscriptions, e.g., enterprise cloud instances [2]. The application performance is often observed to scale sublinearly with the number of cores per VM and the number of VMs due to the limited parallelism of applications and other resource bottlenecks, e.g., network. It is of paramount (economical) interest to answer the question of how to make use of inexpensive VMs to host latency-sensitive applications, e.g., web search, the user experience of which can easily be impaired by high latency.

With the aim to get more capacity out of meager instances, opportunistic strategies [3, 1, 4, 5] have been developed to sniff the performance heterogeneity of theoretically identical instances, i.e., the root causes and the patterns of time-varying capacities. On the one hand, one can try to select VMs that outperform others in conjunction with VM scaling [3, 5]; however, due to the overhead of starting up new VMs and the constraint of billing periods, such actuation is restricted to a macro granularity, e.g., one hour, and may miss the opportunity to capture finer-scale capacity variations. On the other hand, one can perform heterogeneity-aware optimization strategies at a micro time scale, such as load balancing [4] and CPU throttling [1], relying on detailed profiling of the application on each deployed VM.

Another recent strategy to minimize latency is query cloning, which has been shown to be effective in the context of web services [6] and big data processing platforms [7] where the capacity variability is high, yet without requiring intrusive probing of the VMs' states. Upon the arrival of a query, multiple clones are spawned on different servers, and the results are returned immediately after the first clone completes. The effectiveness of cloning depends on the overhead caused by processing additional clones and the possibility of canceling remaining unfinished clones. When the outstanding clones can be canceled immediately after the first clone completes, one can use a larger number of clones to further reduce the latency. In a cloud setting, query cloning opens up new doors in terms of cost effectiveness: is it cheaper to purchase a large number of wimpy VMs and replicate requests to achieve the latency target, or a small number of brawny VMs?

In this paper, we address the challenging question of how to best host latency sensitive applications with dynamic workload patterns and dominated by read only workload, using inexpensive wimpy VMs in the cloud. We advocate to simultaneously scale VM provisioning on the macro scale and query cloning on the micro scale, where the former handles time-varying load, and the latter tackles the transient capacity variability without intrusive profiling. We argue that it is often more economical to allocate a large number of wimpy instances and leverage the power of query redundancy, especially when the marginal cost of additional capacity is high. In other words, the sweet spot for simultaneously scaling wimpy VMs and queries is when doubling resource provisioning, in terms of cores per VM or number of VMs, does not improve the latency by a factor of two, whereas still bearing twice the cost. To exploit such conditions, we develop a set of analytical models and a dual elasticity controller, *DuoScale*, and conduct extensive evaluations on trace-driven simulations whose arrivals and processing times are based on measurements from real applications in the cloud. We aim to answer the following specific questions:

- Is there a performance gain in scaling VM provisioning and query clones simultaneously? And if so, what are the optimal levels to meet a given latency target?
- Is it more economical to use dual scaling on small cheap instances or resource scaling on big instances?
- How to sustain a stringent latency target by scaling both resources and workloads, when encountering dynamic workloads?

We first derive analytical models that incorporate two clone canceling policies: (i) early canceling of remaining unfinished clones with some overhead and (ii) leaving the clones to complete without canceling. Using the derived closed-form expressions for the average latency, we can efficiently explore the design space of different arrival patterns, VM types, number of instances, and query replication levels, so as to achieve the target latency at a minimum cost. Secondly, we develop a model-driven elasticity controller, DuoScale, which dynamically determines the number of VMs at macro timescales, e.g., one hour windows, and tunes the replication levels at micro timescales, e.g., 5 minute windows. Extensive evaluation results from trace-driven simulations and a deployment of the MediaWiki application in the cloud show that scaling *both* VMs and queries simultaneously not only achieves the target latency effectively, but also makes wimpy instances perform better by exploiting their performance variability.

Our contributions are the following: (i) a novel and cost-effective dual scaling strategy that is particularly suitable to meet stringent latency targets using wimpy VM instances; (ii) a set of closed-form results to approximate the latency for any given VM provisioning and workload redundancy level under different clone canceling policies; (iii) extensive evaluations on trace-driven simulations and an application deployment.

II. Motivation

The most common question when hosting services in the public or private cloud is how many VMs of which type are needed. The two key factors affecting this decision are the deployment cost, which is to be kept at a minimum, and the latency delivered, which must meet a certain quality target.

To demonstrate the complexities and challenges in achieving the target latency by resource scaling, we deploy the MediaWiki application in our private cloud. The query target latency set to 120 milliseconds. To serve a load of 10 requests per second, we use two types of VMs, namely wimpy and brawny, which consist of 2 and 4 virtual cores, with 2 and 4 GB RAM, respectively. We progressively increase the number of instances and measure the average latency. Fig. 1 summarizes the results for both instance types. The best latency by scaling wimpy instances is around 138 ms, whereas scaling brawny instances results in an average latency as low as 120 seconds. One may thus conclude that using a sufficient number of brawny instances, i.e., 5, one can achieve the target latency

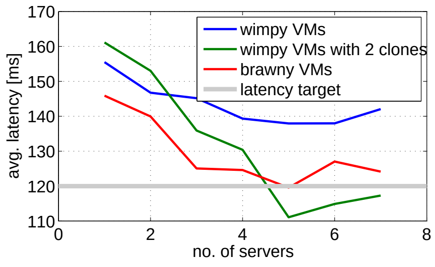


Fig. 1: The average latency of hosting MediaWiki in cloud under three scaling strategies: (i) adjusting wimpy VMs only, (ii) adjusting brawny VMs only, and (iii) adjusting wimpy VMs and cloning requests by a factor of two.

of 120 ms, whereas even a high number of wimpy instances fails to achieve such a target.

Inspired by the recent proposal of query replication for latency curtailment, we duplicate every query upon its arrival and dispatch the original and cloned query to different servers. We return the queries to users as soon as the first clone returns and leave the other clone to continue and complete its execution. The detailed implementation is described in Section VI. As shown in Fig 1, a query replication factor of two, i.e., the original query plus a clone, on a sufficiently large number of wimpy instances, i.e., 5 VMs, can achieve an average latency as low as 115 ms, which is lower than the target and better than the best performance achieved with brawny instances. We also note that when the number of wimpy instances is low, 2 instances or less, replicating queries results in a latency worse than the wimpy instances without query replication. This is due to the extra processing overhead of replicated queries, and indicates that enabling query replication without sufficient resources can result in more harm than performance advantages.

Overall, to achieve the latency target of 120 ms at the load of 10 requests per second there are two alternatives: (i) use five brawny instances, or (ii) use five wimpy instances with a query replication factor of two. The cost associated with those alternatives is the next important criteria to consider. Since according to standard pricing conventions stronger instances are more expensive, it is definitely cheaper to rent five wimpy instances than five brawny instances.

The key message derived from this example is that scaling both resources and workload simultaneously can maintain a stringent latency target in a cost-effective fashion, compared to a solution consisting solely of resource scaling, especially when there exists capacity variability across VM instances. Also, the enabling condition behind this observation is that vertical or horizontal resource scaling, i.e., number of cores per VM or number of VMs, often provides sublinear performance improvement. The immediate challenge in addressing double elasticity is how to determine the optimal resource provisioning and query replication according to the arriving workloads and the latency target. This entails explicitly considering the overheads of resource provisioning and cloning, as well as the

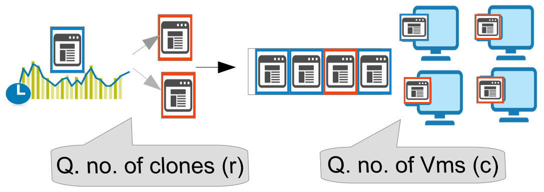


Fig. 2: Schematics of systems capable of dual scaling VMs and query clones. Queries with time-varying arrival patterns are first cloned and then sent to be executed by a pool of VMs.

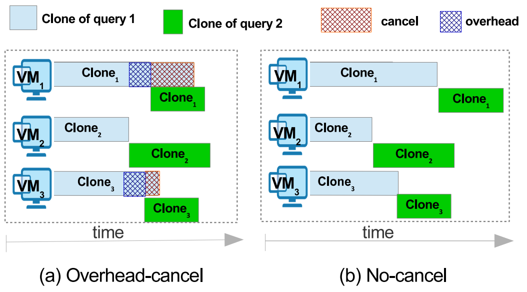


Fig. 3: An example of applying two different clone canceling policies on queries with replication level $r = 3$. Clone two C_2 of the first query is the fastest and clone one C_1 and three C_3 either are terminated or continue, subject to the canceling policy.

multiple VM types, and their price and performance characteristics, so as to achieve the most cost-effective solution.

III. Dual Elasticity on VMs and Queries

This section formally introduces the key aspects to consider in the light of dual scaling. We examine a service provider that utilizes public or private cloud resources to deploy an interactive service, with a particular focus on read-only workloads. The provider is thus concerned with the cost incurred in running this service, as well as with the quality of service experienced by the service users. The latter is reflected in a target $\bar{R}$ for the average latency.

A key feature of this system is that it is subject to a time-varying demand pattern, i.e., the arrival rate of queries $\lambda(t)$ varies with time t. At any given time t, the system consists of $C(t)$ VMs, each with service rate of μ queries per second. As we focus on read-only workloads for interactive services, we further assume that all VMs are *homogeneous* and capable of executing all queries. Queries arrive at a central queue, where they can be cloned before being dispatched to the next available server[1], as shown in Fig. 2. The central queue also takes care of returning the response to the client once the first clone completes, and initiates the canceling of outstanding clones if such a policy is in place.

To maintain the latency target, the system is able to scale (i) the number of provisioned VMs, $C(t)$, and (ii) the number of clones, or query replication level, $r(t)$, according to the (estimated) arrival rates. We explain the design considerations regarding these two control variables in the following. For simplicity we drop the index t unless it is explicitly required.

On-demand VM Provisioning. In theory, VMs in the cloud can be provisioned and decommissioned according to the rise and fall of arrival rates; however, the typical practice is to avoid frequent VM scalings due to several practical concerns. On the one hand, scaling up the number of VMs entails delays [8] caused by VM image retrieval, OS booting, application installation, and cross configuration with other servers. On the other hand, even if VMs can be decommissioned with little or no overhead delay, terminating VMs before the end of their billing period, which is typically in the order of one hour, leads to the waste of already paid resources. As a result, VM scaling tends to be actuated at a coarse granularity to amortize the overhead of start-up delay and avoid the financial waste stemming from the early termination of VMs.

Query Replication. Cloning queries has been proposed as a solution to mitigate slow execution, either reactively after experiencing long delays, or proactively upon the arrival of queries [7]. All clones are sent to different VMs so as to best take advantage of the variability across VMs, i.e., to increase the probability of a clone being executed on a "fast" VM. For each query, as soon as one of its clones completes, the result is returned to the user. Here, we particularly focus on proactive query replication, i.e., every query is replicated upon arrival.

To deal with the additional load introduced by proactive cloning, a canceling mechanism can be introduced to drop the outstanding clones after the completion of the first one. As illustrated in Fig. 3, there are two main canceling policies: (i) overhead cancel, where outstanding clones can be removed with a certain delay overhead, as in Fig. 3(a), where new queries at VM 1 and 3 can start only after the canceling overhead delay marked as dashed blue rectangles; and (ii) no cancel, where outstanding clones continue their execution uninterrupted, as in Fig. 3(b). Canceling without overhead would clearly be preferable, but some canceling overhead is unavoidable in practice and might even be large enough to make no-canceling a better option. All in all, the optimal replication level depends not only on the current load and capacity, but also on the overhead of cloning and canceling queries.

IV. Models and Analysis

In this section, we analytically derive the average query latency given the number of VMs C and clones r, under the two clone canceling policies described in the previous section. In contrast to standard queueing systems, the introduction of clones modifies the arrival process, as all clones of a query arrive as a single batch. More importantly, the service process

[1]We follow the convention of referring to the original request as query and to the requests executed at the server as clones.

and waiting times are affected by the canceling policy, as depicted in Fig. 3. Consequently, one needs to derive the *query* service time, even if the *clone* service times follow known statistical distributions, before obtaining the query response time. In the following we provide approximations for both overhead cancel and no cancel policies. We assume Poisson query arrivals with rate λ and exponential *clone* service times with mean $1/\mu$. Also, the number of clones r is at most equal to the number of VMs C as more clones would be redundant.

To derive the average query response time under the overhead cancel and no cancel policies, we first derive the stationary distribution of the number of queries in the system (π), the mean query waiting time (W_q), and the mean query service time (S), and then compute the response time as the sum of W_q and S.

A. *Overhead cancel*

In this section we propose an approximated method that allows us to obtain closed form formulas for the mean response time and further ease the integration of the model in the DuoScale controller. Let us first consider the case of canceling without overhead, where all outstanding clones of a query are canceled as soon as the first clone completes service. The key observation to analyze this setup is that at any time there can be at most K queries in service, with $K = \lceil \frac{C}{r} \rceil$. If at most $K-1$ queries are present, they all execute with all their clones. Instead, the K-th query, i.e., the youngest, executes with only $h = C \mod r$ clones. This configuration holds at all times due to the synchronization introduced by the canceling mechanism, which removes all clones of a query once it finishes, allowing $r-h$ clones of the youngest query to join service and a new query to start service with h clones. In case the youngest query is the one that terminates, it frees h service spots and allows a new query to start service with h clones.

We now introduce the overhead as a fraction α of the mean processing time, thus α/μ. We assume this overhead is constant if there are at least two clones in service, or zero if a single clone is in execution as canceling is not required in this case. The mean processing time with $j \geq 2$ clones in service is thus $1/(j\mu) + \alpha/\mu$, where the first term is the service time without overhead, given by the minimum of j exponential random variables with mean $1/\mu$. The inverse of this expression is the clone processing rate $\hat{\mu}_j$ with j clones in service, which is equal to

$$\hat{\mu}_j = \begin{cases} \mu, \ j=1, \\ \frac{j\mu}{1+j\alpha}, \ 2 \leq j \leq r. \end{cases} \tag{1}$$

With these definitions we obtain the long-run probability $\hat{\pi}_j$ that there are j queries in the system, in the following result.

Lemma 1. *If $h \geq 2$, the stationary distribution of the number of queries in the system with overhead-cancel, $\{\hat{\pi}_j\}_{j\geq 0}$, is*

$$\hat{\pi}_j = \begin{cases} \left(\frac{\lambda(1+r\alpha)}{r\mu}\right)^j \frac{1}{j!}\pi_0, & 1 \leq j \leq K-1, \\ \left(\frac{\lambda(1+r\alpha)}{r\mu}\right)^{K-1} \frac{\hat{\rho}^{j-K+1}}{(K-1)!}\pi_0, & j \geq K, \end{cases} \tag{2}$$

where $\hat{\pi}_0$ is given by

$$\hat{\pi}_0 = \left(1 + \sum_{j=1}^{K-1} \left(\frac{\lambda(1+r\alpha)}{r\mu}\right)^j \frac{1}{j!} + \right. \tag{3}$$

$$\left. \left(\frac{\lambda(1+r\alpha)}{r\mu}\right)^{K-1} \frac{1}{(K-1)!}\frac{\hat{\rho}}{1-\hat{\rho}}\right)^{-1}, \tag{4}$$

and

$$\hat{\rho} = \frac{\lambda(1+r\alpha)(1+h\alpha)}{C\mu + K\alpha h r\mu}. \tag{5}$$

Proof. We setup a birth-and-death (BD) process [9] that counts the number of *queries* (one query accounting for all its clones) in the system. The state space is thus the set of natural numbers, and the birth rates are equal to λ in every state, marking the arrival of new queries. For the death rate γ_k in state $k \geq 1$ we use (1) and assume $r \geq 2$ to obtain

$$\gamma_k = \begin{cases} \frac{kr\mu}{1+r\alpha}, \ 1 \leq k \leq K-1, \\ \frac{(K-1)r\mu}{1+r\alpha} + \mu, \ k \geq K, h=1, \\ \frac{(K-1)r\mu}{1+r\alpha} + \frac{h\mu}{1+h\alpha}, \ k \geq K, h \geq 2. \end{cases} \tag{6}$$

In the first case up to $K-1$ queries execute with all their r clones. In the second and third cases the K-th query executes with h clones only, and the canceling overhead is introduced if $h \geq 2$. We thus set up and solve the balance equations for the BD process to obtain (2) and (4), where $\hat{\rho}$ is the ratio between the birth and death rates for all states $k \geq K$, thus

$$\hat{\rho} = \frac{\lambda}{\frac{(K-1)r\mu}{1+r\alpha} + \frac{h\mu}{1+h\alpha}},$$

from which we obtain (5). □

Lemma 1 holds if the system is stable [9], i.e., if and only if $\hat{\rho} < 1$. From Lemma 1 we obtain the mean query queue length and applying Little's Law we find the query mean waiting time as follows.

Corollary 1. *If $h \geq 2$, the query mean waiting time is given by*

$$\hat{W}_q = \hat{\pi}_{K-1}\frac{\hat{\rho}}{(1-\hat{\rho})^2}\frac{(1+r\alpha)(1+h\alpha)}{C\mu + K\alpha h r\mu}. \tag{7}$$

The analysis above assumes that the query in front of the queue can start service only after a query in service has completed service *and has canceled all its clones*. However, in Fig. 3 we see that one clone in the queue can start service as soon as the first clone of the query in service completes, and the remaining $h-1$ clones join once the canceling step has been performed. To compensate for this additional delay in the waiting time, we allow all h clones to start service simultaneously, skipping the period where a single clone executes on its own. Thus, all clones start at the same time as the first one, but they all have to wait until after the canceling step is performed. With this approximation we obtain the query mean service time in the following result.

Lemma 2. *If $h \geq 2$, the query mean service time is*

$$S = \frac{1}{r\mu}\left(1 + \frac{\hat{\pi}_{K-1}}{1-\hat{\rho}}\frac{(r-h)(1+\alpha r)}{C+K\alpha hr}\right). \tag{8}$$

Proof. We consider two cases: (i) the query starts service with all its clones, with probability $\sum_{j=0}^{K-2}\pi_j$ due to the PASTA property [10], and the mean service time is $1/r\mu$ as the first of the r clones to finish determines the completion time; (ii) the query starts service with h clones with probability $\sum_{j\geq K-1}\pi_j$ as it finds h idle servers with probability π_{K-1}, or it joins the queue with probability $\sum_{j\geq K}\pi_j$, eventually starting service with h clones. If one of the first h clones completes service before the clone of any other query in service, which from (6) occurs with probability $(h\mu/(1+h\alpha))/\gamma_K$, the mean service time is $1/h\mu$. Instead, if the first clone to complete service is from a different query, with probability $1-(h\mu/(1+h\alpha))/\gamma_K$, the mean service time is $1/r\mu$ due to the memoryless property of the exponential distribution. Putting everything together we obtain

$$S = \frac{1}{r\mu}\sum_{j=0}^{K-2}\hat{\pi}_j + \left(\frac{1}{h\mu}\frac{\frac{h\mu}{1+h\alpha}}{\gamma_K} + \frac{1}{r\mu}\left(1-\frac{\frac{h\mu}{1+h\alpha}}{\gamma_K}\right)\right)\sum_{j\geq K-1}\hat{\pi}_j,$$

where we replace (2) to obtain (8). □

Adding the mean waiting time in (7) to the mean service time in (8) we obtain the approximate query mean response time $\hat{R} = \hat{W}_q + \hat{S}$ in the system with canceling overhead.

The results for the case $h = 1$ can be obtained similarly, and we summarize them in the following result.

Lemma 3. *If $h = 1$, the stationary distribution of the number of queries is as in Lemma 1, but the traffic coefficient is*

$$\hat{\rho} = \frac{\lambda(1+r\alpha)}{\mu(C+r\alpha)}.$$

The mean waiting time is given by

$$\hat{W}_q = \hat{\pi}_{K-1}\frac{\hat{\rho}}{(1-\hat{\rho})^2}\frac{1+r\alpha}{\mu(C+r\alpha)},$$

and the mean service time is given by

$$S = \frac{1}{r\mu}\left(1 + \frac{\hat{\pi}_{K-1}}{1-\hat{\rho}}\frac{r-h+\alpha r(r-1)}{C+r\alpha}\right).$$

B. No cancel

We now consider the policy where all clones execute until completion without being canceled. This system is difficult to analyze exactly as it requires keeping track of single clones instead of queries. We thus opt for an approximate analysis where clones are modeled separately but their arrival in batches, caused by cloning, is captured through the impact that these batch arrivals have on the variability of the arrival process. This enables us to approximate the request mean waiting time.

As we look at the level of clones, we model the system as an M/M/C queue with arrival rate $r\lambda$. From the analysis of the M/M/C, and defining $\rho = r\lambda/C\mu$, we readily obtain [9] the stationary distribution of the number of clones $\{\pi_j\}_{j\geq 0}$ as

$$\pi_j = \begin{cases} \frac{(C\rho)^j}{j!}\pi_0, & 1 \leq j \leq C-1, \\ \frac{C^C\rho^j}{C!}\pi_0, & j \geq C, \end{cases} \tag{9}$$

where π_0 is given by

$$\pi_0 = \left(\sum_{j=0}^{C-1}\frac{(C\rho)^j}{j!} + \frac{(C\rho)^C}{C!}\frac{1}{1-\rho}\right)^{-1}.$$

This stationary distribution exists if the system is stable [9], i.e., if and only if $\rho < 1$. Thus, introducing r clones without canceling reduces the maximum sustainable arrival rate by a factor r. The mean waiting time in this queue is given by

$$\pi_0\frac{(C\rho)^C}{C!}\frac{1}{1-\rho}\frac{1}{C\mu - r\lambda}.$$

However, since the clones arrive in batches we approximate the query mean waiting time via the Allen-Cunneen approximation [11]

$$W_q = \pi_0\frac{(C\rho)^C}{C!}\frac{1}{1-\rho}\frac{1}{C\mu-r\lambda}\frac{r+1}{2}, \tag{10}$$

where the numerator of the factor on the right is the sum of the coefficients of variation (CV) of the clone inter-arrival times (r) and the service times (1). The service time CV is one because the clone service times are exponentially distributed. Instead, the inter-arrival time CV is r times that of the *query* inter-arrival times, which is one as queries arrive as a Poisson process. This approximation is well-known [11] to capture the impact of batch arrivals on mean waiting times. We now obtain the query mean service time in the following result.

Lemma 4. *The query mean service time in the system without canceling is given by*

$$S = g(r,r)\sum_{j=0}^{C-r}\pi_j + \sum_{j=C-r+1}^{C-2}\pi_j g(r, C-j) + g(r,1)\sum_{j\geq C-1}\pi_j, \tag{11}$$

where $g(i,r)$ is the mean request service time given that the request starts service with i out of r clones, and is given by

$$g(r,i) = \frac{i}{C}\frac{1}{i\mu} + \frac{C-i}{C}g(i+1,r). \tag{12}$$

Proof. First, to obtain (12) we consider two cases: either the initial i clones finish service before any other clone in service, with probability i/C, causing a mean service time of $1/i\mu$; or any other clone in service finishes first, with probability $(C-i)/C$, and the mean service time is given by $g(i+1,r)$. Once all the $g(i,r)$ terms are known, we can obtain (11) by relying on the PASTA property. Here we consider three cases: (i) the request sees up to $C-r$ clones in service and starts service with all its clones, requiring mean service time $g(r,r)$; (ii) the request sees $2 \leq j \leq r-1$ idle servers, thus starting with j clones initially in service and requiring a mean service time $g(r,j)$; (iii) the request finds either one or zero idle servers, thus starting service with one clone and requiring mean service time $g(r, 1)$. $\square$

As before, we add the mean waiting time in (10) to the mean service time in (11) to obtain the query mean response time in the system without canceling.

C. Model Validation

Here we validate the derived analysis via trace-driven simulation, the details of which can be found in Section VI. The aim is to see how such an approximate closed form analysis can accurately capture the complex system dynamics and latency. For both clone policies, we evaluate two particular scenarios: (i) synthetic case: Poisson arrivals with an average rate $\lambda = 1$ and exponentially distributed clone processing times with an average rate of $\mu = 1$; and (ii) empirical case: empirical inter-arrival times with an average rate $\lambda = 1.5$ and empirical clone processing times with an average rate $\mu = 1$. We set a canceling overhead of 10%. The comparison with the synthetic case is to demonstrate the accuracy of our approximation for systems that have exactly the same workload inputs, whereas the comparison with the empirical case is to evaluate the effectiveness of the derived analysis in guiding the choices of optimal servers and clone levels.

To obtain results for realistic arrival patterns, we rely on the publicly available Wikipedia trace [12]. We focus on a time window where inter-arrival times are statistically stationary. For the empirical query processing time results, we execute MediaWiki, an open source implementation of Wikipedia, on our cloud testbed (details can be found in Section VI). We note that the empirical inter-arrival times show a higher variability than Poisson arrivals, and the empirical processing times have a CV around 0.9, which is slightly lower than for the exponential case. We summarize the validation results for both models in Fig. 4, where we present the average latency with respect to different numbers of VMs and clones. We note that certain combinations of C and r are not shown due to violations of the system stability conditions. Moreover, we do not evaluate cases where the number of clones is greater than the number of VMs, i.e., $r > C$, as this setting counters the idea of sending clones to different VMs.

The general observations on model accuracy from Fig. 4 are as follows. (i) The derived analysis is accurate for the synthetic traces, even though it is approximate. (ii) The model tends to be over-optimistic in predicting latency for the empirical case. This is due to the fact that empirical service times of clones have lower CV than the exponential times. The higher the variability of service times is, the better the potential performance gains from adding clones. (iii) Adding query clones indeed lowers the latency, provided there are sufficient VMs. For example, two clones have better performance than one clone when the number of VMs is greater than 3 (4) for overhead cancel (no cancel). (iv) The prediction error grows with the number of clones.

When zooming in on each cloning policy individually, we can see that for overhead cancel (see Fig. 4(a)-(b)), the

TABLE I: Estimated latency used by the `predictor` for the two canceling policies.

Clone canceling policy	$E[R] = W_q + S$
Overhead cancel	Eq. (7) + Eq. (8)
No cancel	Eq. (10) + Eq. (11)

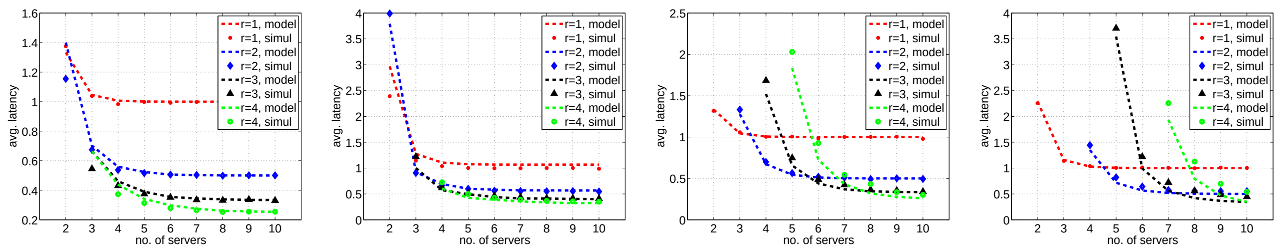
resulting average errors are 3% and 1% for the synthetic and empirical case, respectively. When $C \geq 3$, the lowest average latency is achieved by creating 4 clones under both synthetic and empirical distribution. As for the no cancel case (see Fig. 4(c)-(d)), the resulting average errors are 7% and 12% for the synthetic and empirical case, respectively. Given its approximated nature, our model indeed returns remarkably accurate predictions. In contrast to overhead cancel, the performance gain realized by adding clones is very limited here due to the higher processing load. When using clones without the option of canceling, one shall cautiously provide sufficient VMs to avoid overloads. For example, one shall only consider using 2 (3) clones when there more than 5 (7) VMs; otherwise the overhead of processing clones outweighs the performance gain, compared to the case without cloning, i.e., $r = 1$.

In a nutshell, our extensive evaluation strongly supports that increasing the VM provisioning level to accommodate additional load to process clones can reach a very stringent latency target which is lower than the average processing time without cloning. The results also demonstrate the complex performance dynamics across different types and numbers of VMs, numbers of clones and canceling policies.

V. Dual Elasticity Controller

We develop a model-driven dual elasticity controller, DuoScale, which can dynamically and simultaneously scale the VM provisioning and query cloning. Due to the billing constraints and the VM scaling overhead, DuoScale actuates at two different granularities: VM scaling in macro windows and clone adjustment in micro windows. DuoScale consists of four main components, `monitor`, `predictor`, `clone actuator` and `VM actuator`, depicted in Fig. 5. The `monitor` collects inter-arrival times, processing times, and latencies at every micro window. Based on them, the `predictor` forecasts the workload and determines the optimal number of VMs/clones at every micro and macro window. The two actuators execute the decisions of the `predictor`.

At every macro window, the `predictor` forecasts the arrival rate for the next macro window and determines the minimum number of VMs that can achieve the target latency subject to the clone canceling policy, according to the predicted latency summarized in Table I. To accommodate the errors in predicting the arrival rates, the `predictor` looks T micro windows ahead and selects the maximum of $\{\lambda(k), t+1 \leq k \leq t+T\}$ as the estimated arrival rate for the next macro window. Hence, DuoScale tends to be conservative in allocating VMs and this spare capacity can be exploited to accommodate additional clones. We note that there is a large selection of time series prediction tools, e.g., [2], which can



(a) Overhead cancel: syn (b) Overhead cancel: emp (c) No cancel: syn (d) No cancel: emp

Fig. 4: Model validation for clone canceling policies on synthetic traces ($\lambda = 1$, and $\mu = 1$) and empirical traces ($\lambda = 1.5$, and $\mu = 1$).

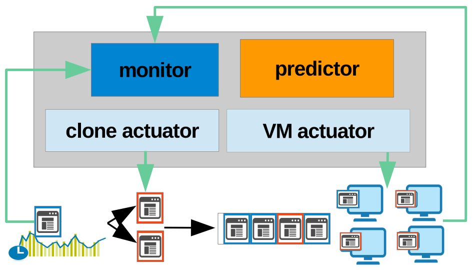
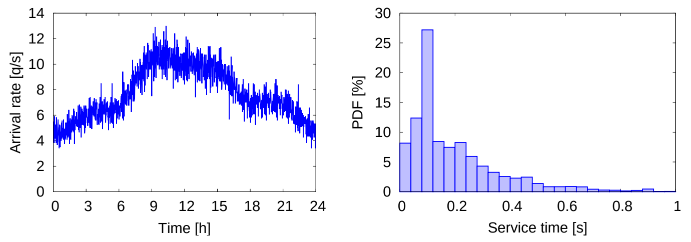


Fig. 5: Architecture of DuoScale.



(a) Query arrival pattern (b) Service time distribution

Fig. 6: 24-hour Wikipedia arrival trace empirical clone service time distribution normalized to mean arrival rate 7.5 req/sec and service rate 5 clones/sec.

effectively predict the average arrival rates. After obtaining the workload estimates, the `predictor` makes use of the closed formulas summarized in Table I to sweep through all possible combinations of VM and clone numbers, and returns the combination that can achieve the target latency using the minimum cost, i.e., the minimum number of VMs. At every micro window, the `predictor` forecasts the arrival rate for the next micro window and computes the optimal number of clones according to Table I, with the number of VMs fixed by the macro window.

In this paper, we consider one hour macro windows and 5 minute micro windows in most experiments. Every hour, the `predictor` forecasts the average 5-minute arrival rates for the next $T = 12$ intervals and chooses the maximum value as the prediction for the next hour. Every 5 minutes, the `predictor` adjusts the clone levels according to the models. We note that the accuracy of the arrival rate prediction and the window size can affect the effectiveness of DuoScale, but the selection of optimal forecast methods and parameters is out of the scope of this paper.

VI. Evaluation

In this section we present the average response times and the cost of applying DuoScale on wimpy VMs, using trace-driven simulations and experiments on a web application deployed in the cloud. We compare DuoScale against two baseline solutions that scale either wimpy or brawny VMs. We first explain the setup, then summarize the trace simulation results, and conclude with the test-bed results.

A. Simulation Setup

We use MediaWiki, the open source platform used to run the Wikipedia website, as a representative latency-sensitive web application in the cloud [12] and evaluate DuoScale with an in-house discrete event simulator based on Omnet++. The inputs for the simulator are inter-arrival and processing times that can be drawn from a statistical distribution or empirical traces. We particularly focus on trace-driven simulation, whose query arrivals are scaled down from the trace collected at the real Wikipedia site on September 19, 2007 [13]. In terms of service times, we use both the exponential distribution and an empirical distribution collected by deploying MediaWiki on our cloud testbed. Fig. 6 shows the arrival rate in 5-minute intervals along one day and the distribution of the service times. We consider two VM types: (i) wimpy VMs with an average clone service rate of $\mu = 5$ clones/sec, and (ii) brawny VMs with an average clone service rate of $\mu = 7.5$ clones/sec. We assume the price to rent brawny VMs to be twice that of wimpy VMs, proportional to their resources, as is common practice in public cloud providers. However, the service rate of brawny VMs is only 1.5 times that of wimpy VMs, following observations made on extensive experimentation on our testbed. The 1.5 gain is in fact an upper bound on the gains observed in the testbed, which are around 20%. Moreover, we normalize the traces such that the average arrival rate is $\lambda = 7.5$ req/sec.

TABLE II: Comparisons among two VM scale only solutions (WV scale and BV scale) and DuoScale on three canceling policies: achieving the target latency of 140 ms under exponential and empirical clone processing times.

Target = 140 ms	Avg. VMs [#]	Normalized cost	Avg. clones [#]	Exponential		Empirical	
				Resp. time [ms]	Violations [%]	Resp. time [ms]	Violations [%]
WV scale	8.17	2.13	1.00	200.00	100	200.00	100
BV scale	3.79	1.98	1.00	134.5	21.50	134.1	22.19
DuoScale (No cancel)	6.38	1.66	2.01	114.2	2.30	124.3	6.48
DuoScale (10% overhead)	4.54	1.18	3.46	86.7	0.00	112.8	2.02
DuoScale (0% overhead)	3.83	1.00	3.87	88.6	0.70	134.7	34.56

B. Effectiveness of DuoScale

We apply DuoScale to achieve a latency target of 140 ms per query on a one-day Wikipedia trace, dynamically adapting the number of VMs every hour and the clone levels every 5 minutes. We also include results from the two baselines that every hour scale wimpy VMs only (WV scale) or brawny VMs only (BV scale). To make the comparison fair, WV scale and BV scale use the same workload inputs as DuoScale and the same model for scaling, but the number of clones is fixed to one in both cases. Moreover, when the model predicts there is no feasible solution, e.g., using WV scale to achieve the target of 140 ms, we choose the number of servers such that the marginal gain in response time of adding one server is less than 0.1%.

Table II summarizes the results for WV scale, BV scale, and DuoScale with the two canceling policies, namely no cancel, and overhead cancel with the overhead being 0 and 10%. We present the average number of VMs per hour, the average number of clones per query, and the total cost, normalized by the case of DuoScale with 0 canceling overhead. We also show the average query response times and the percentage of violations (micro windows where the average latency exceeds the target) for both exponential and empirical cases. We note that as the `predictor` of DuoScale only uses the clone average service time as input, it makes the same VM provisioning and query cloning decisions for both exponential and empirical cases. We thus do not present separate values of cost, numbers of servers and number of clones for the exponential and empirical cases.

General observations. Under all canceling policies, DuoScale and BV scale are able to maintain the target latency, whereas WV scale is not. More importantly, DuoScale achieves the best mean response times at the lowest cost, i.e., it features costs savings between 20 to 50% compared to BV scale and VM scale. DuoScale uses the smallest number of VMs (3.83) and the largest number of clones (3.87) under 0 canceling overhead, whereas it uses the largest number of VMs (6.38) and the smallest number of clones (2.01) under no cancel. Under no cancel, DuoScale allocates almost twice as many VMs compared to overhead cancel (particularly 0 overhead) and introduces just enough spare capacity to exploit cloning. These numbers reflect how DuoScale chooses the best combination of VMs and clones to achieve the latency target, and how DuoScale clones more aggressively under overhead cancel and more conservatively under no cancel due to the different associated costs of introducing clones. Looking into the performance at the micro window level, DuoScale combined with no-cancel gives the most robust results, with only a small fraction of micro windows, around 5%, displaying violations. Although BV scale can keep the overall average latency below the target, i.e., 0.67 seconds, it causes more (22%) violations and a higher cost.

Exponential v.s. empirical. DuoScale performs better under exponentially distributed service times, offering lower average latencies and significantly less violations, across all canceling policies. This is because the variability in the empirical service times is lower than in exponentially distributed ones, leading the `predictor` to overestimate the benefits of clones especially for the 0 canceling overhead case.

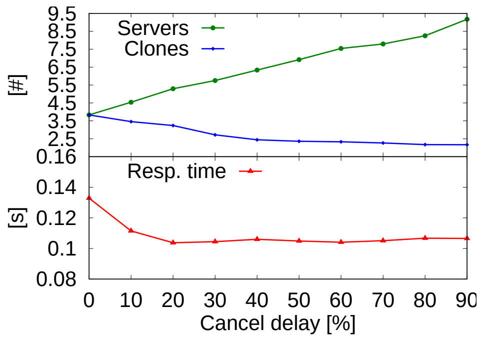


Fig. 7: Impact of different fractions of canceling overhead on DuoScale: number of VMs/clones and average latency.

C. Impact of the Canceling Overhead

To study the impact of the canceling overhead, we evaluate DuoScale under canceling overheads ranging from 10 to 90% with empirical service times. Fig. 7 shows, in the top row, the number of VMs and clones as decided by DuoScale and, in the bottom row, the average latency. Remarkably, DuoScale is able to accurately capture the impact of the overhead and correspondingly scales up (down) the number of VMs (clones) as the overhead increases. Recall from Table II that no cancel requires an average provisioning of 6.38 VMs. Thus, when the average number of servers needed for overhead cancel exceeds 6.38 VMs, it is actually preferable to let all clones continue their execution, instead of canceling them. From Fig. 7 we observe that overhead cancel offers a reduction in the number of VMs when the overhead is less than 45%. The `predictor` of DuoScale thus offers a mechanism to evaluate whether canceling should be introduced considering the associated overhead and potential gains.

D. Cloud testbed

We evaluate the cost-performance effectiveness of the DuoScale prototype on a web service cluster, namely MediaWiki, hosted on our private cloud. The objective is to deliver a request within 140 ms against dynamic loads with a minimum amount of provisioning cost. Similar to the previous subsection, we compare DuoScale against the solution that only scales brawny VMs (BV scale). The `predictor` of DuoScale uses the no cancel model, as we do not cancel clones after starting the execution.

Testbed Setup We consider two kinds of VMs: wimpy instances, each equipped with two virtual cores and 2 GB RAM; and brawny instances equipped with 4 virtual cores and 4 GB RAM. The cost per brawny instance is twice that of a wimpy instance, following a linear pricing scheme adopted in most current cloud offerings. MediaWiki is a latency-sensitive web application composed of Apache (v2.4.7) plus PHP (v5.5.9) as application server, memcached server, and MySQL (v5.5.40) as DB server. The requests are generated with httperf [14], an open-loop workload generator, following the time-varying arrival rate in Fig. 8(a). To expedite the experiment time, we use 15 minutes as the length for macro windows and 1 minute for micro windows. For a fair comparison, we use the `predictor` of DuoScale to determine the number of VMs for both VM scaling strategies.

The normalized cost. We summarize the number of servers used by both strategies in Fig. 8(b). Clearly, DuoScale consistently uses fewer servers than BV scale along the experiment duration, and the number of VMs used by DuoScale closely follows the arrival rates. As a result, DuoScale results in a cost reduction of slightly more than 50% compared to BV scale, as shown in Fig. 8(c). We note that brawny VMs are less attractive here than in the simulation results presented earlier, because the performance of brawny VMs is only 1.2X better than wimpy VMs in this testbed scenario whereas we assumed a performance factor of 1.5 in Section VI-B. When such a performance factor is close to 2, we expect that BV scale can provide a better cost-performance ratio than DuoScale.

Average latency. Fig. 8(d) presents the average latency: both strategies fulfill the latency requirements of 140 ms in a conservative manner. This can be explained by the conservative prediction of arrival rates. DuoScale has the best average latency, i.e., around 88 ms, whereas BV scale offers an average latency around 120 ms. DuoScale thus achieves the best cost-performance ratio in a practical real-life scenario, compared to pure resource elasticity. Our results show that DuoScale is able to adapt to the system dynamics compound with the arrival rates and clone service rates, and choose resource provisioning and query redundancy levels in a cost-effective manner.

VII. Related Work

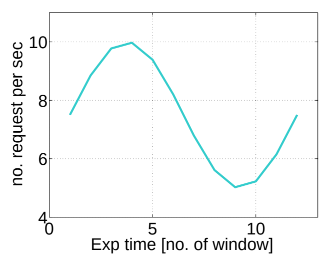
Performance heterogeneity across and within VM types is one of the main topics in prior art, with a recent shift towards meager instances whose resource capacities are very limited. We highlight studies and techniques which are relevant in the context of (i) analyzing the performance variability and (ii) mitigating performance pitfalls by scaling resources and queries.

Performance variability. Measurement studies in the cloud point out that the performance variability of VMs can be grossly classified into (i) longer and (ii) shorter range, depending on their root causes. The former is often attributed to the underlying architecture heterogeneity [3, 15] and providers' policies [16], whereas the latter is often associated with resource contention due to co-located workloads [1] or network interruptions [17], and garbage collection [18]. While [1] uses the transient CPU throttling of micro instances on the host side in the order of tens of seconds, [17] puts forward a token mechanism to allocate network and memory bandwidth to micro instances, in the order of several minutes. One simple way to overcome the longer-term VM variability within the same type is to opportunistically choose VMs with a better performance [3, 5] and drop the under-performing ones, in synchronization with the discrete windows of dynamic VM provisioning. To mitigate the short range variability, several effective techniques that operate in fine time granularity are proposed, e.g., scheduling [4] and injecting idling time [1], but require deep understanding of applications via profiling.

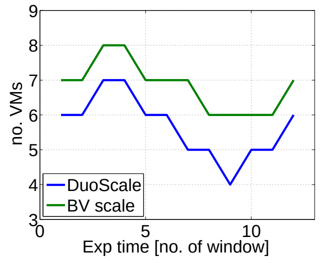
Scaling resources. There is a plethora of studies [19, 20, 21, 22] on elastic resource controllers that determine the number of VMs and their types such that the costs of operation, energy, and performance penalty are minimized. Based on the (predicted) workloads, e.g., number of arriving queries per second, VMs are (de)allocated to achieve the target utilization or latency, considering application start up overheads and the billing cycle of instances. The field data collected from production datacenters, e.g., IBM [2], shows that the average utilization is roughly 20%, leaving plenty of room for additional workloads, including clones.

Query replications. Cloning queries speculatively has been shown to be an effective strategy to improve the response time from conventional web services [6] to recent big data platforms [7, 23], with the implicit assumption of sufficient capacity to accommodate clones. Reactively cloning queries after detecting stragglers adds additional delay [24], while proactive cloning upon arrival demands a higher capacity availability [7]. Being able to cancel the remaining unfinished clones can increase the room to accommodate more clones, increasing the probabilities that queries are served by a fast server [25, 6]. Recently developed queueing models [26, 27, 28, 29] that try to identify the optimal replication levels that achieve the minimum latency assume clones can be canceled without any overhead; with the exception of [30] whose computational overhead hinders its applicability for dynamic workloads, and [31] which focuses on scheduling policies.

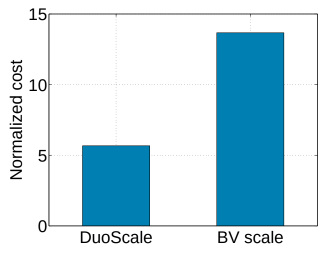
We derive an approximate latency analysis for query replication with both overhead canceling and no canceling. Thanks to its closed-form expression and high accuracy, we are able to efficiently explore the multi dimensional design space, i.e., workload patterns, VM types, VM numbers, and clone num-



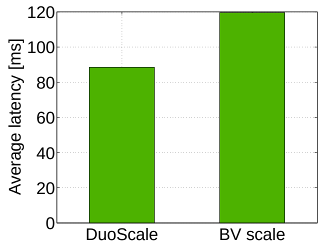
(a)Arrival rates



(b) No. of servers allocated



(c) Total normalized cost



(d) Average latency [ms]

Fig. 8: Applying DuoScale on a MediaWiki cluster hosted on cloud with time-varying arrivals, compared with solutions to scale brawny VMs only.

bers to achieve stringent latency targets using (in)expensive instances in the cloud.

VIII. Conclusion

In this paper, we present DuoScale, a model driven elastic controller, which simultaneously and dynamically scales both VMs and query clones against dynamic loads and capacity variability. The core of DuoScale is a set of analytical models that can accurately predict the average latency under two different clone canceling policies, i.e., overhead cancel and no cancel. By providing slightly more wimpy VMs and making spare room to process clones, DuoScale is able to achieve very stringent latency targets that are even lower than the average processing time of individual wimpy VMs. Extensive results from trace-driven simulations and a prototype on a cloud testbed show that DuoScale is able to achieve the target latency with savings up to 50%, compared to solutions that only scale wimpy or brawny VMs without scaling query clones.

References

[1] J. Wen, L. Lu, G. Casale, and E. Smirni, “Less can be more: Micro-managing vms in amazon EC2,” in *IEEE CLOUD*, 2015, pp. 317–324.

[2] J. Xue, R. Birke, L. Y. Chen, and E. Smirni, “Managing data center tickets: Prediction and active sizing,” in *DSN*, 2016, pp. 335–346.

[3] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, “More for your money: exploiting performance heterogeneity in public clouds,” in *SoCC*, 2012, p. 20.

[4] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, “Bobtail: Avoiding long tails in the cloud,” in *NSDI*, 2013, pp. 329–342.

[5] M. Björkqvist, L. Y. Chen, and W. Binder, “Opportunistic Service Provisioning in the Cloud,” in *IEEE CLOUD*, 2012, pp. 237–244.

[6] J. Dean and L. A. Barroso, “The tail at scale,” *ACM Commun.*, vol. 56, no. 2, pp. 74–80, 2013.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Effective straggler mitigation: Attack of the clones,” in *NSDI*, 2013, pp. 185–198.

[8] M. Mao and M. Humphrey, “A performance study on the VM startup time in the cloud,” in *IEEE CLOUD*, 2012, pp. 423–430.

[9] S. Asmussen, *Applied Probability and Queues*. Springer, 2003.

[10] R. W. Wolff, “Poisson arrivals see time averages,” *Operations Research*, 1982.

[11] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains*. Wiley, 2006.

[12] “Mediawiki,” https://www.mediawiki.org.

[13] G. Urdaneta, G. Pierre, and M. van Steen, “Wikipedia workload analysis for decentralized hosting,” *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[14] “httperf,” http://www.hpl.hp.com/research/linux/httperf.

[15] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, “Runtime measurements in the cloud: observing, analyzing, and reducing variance,” *VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[16] M. Hajjat, R. Liu, Y. Chang, T. E. Ng, and S. Rao, “Application-specific configuration selection in the cloud: impact of provider policy and potential of systematic testing,” in *INFOCOM*, 2015, pp. 873–881.

[17] N. Nasiriani, C. Wang, G. Kesidis, and B. Urgaonkar, “A measurement-based study of effective capacity dynamism on amazon ec2,” in *School of EECS Technical Report No. CSE-16-005*, 2015.

[18] K. Gardner, M. Harchol-Balter, and A. Scheller-Wolf, “A better model for job redundancy: Decoupling server slowdown and job size,” in *IEEE MASCOTS*, 2016.

[19] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, “Dynamic right-sizing for power-proportional data centers,” *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1378–1391, 2013.

[20] Z. Liu, A. Wierman, Y. Chen, B. Razon, and N. Chen, “Data center demand response: avoiding the coincident peak via workload shifting and local generation,” in *Sigmetrics*, 2013, pp. 341–342.

[21] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, “Cloudscale: elastic resource scaling for multi-tenant cloud systems,” in *SoCC*, 2011, p. 5.

[22] L. Mashayekhy, M. M. Nejad, and D. Grosu, “A PTAS mechanism for provisioning and allocation of heteroge-

neous cloud resources," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2386–2399, 2015.

[23] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *SIGCOMM 2015*, 2015, pp. 379–392.

[24] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *PVLDB*, vol. 3, no. 1, pp. 330–339, 2010.

[25] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *CoNEXT*, 2013, pp. 283–294.

[26] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä, "Reducing latency via redundant requests: Exact analysis," in *ACM Sigmetrics*, 2015, pp. 347–360.

[27] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" in *Allerton*, 2013, pp. 731–738.

[28] Z. Qiu and J. F. Pérez, "Enhancing reliability and response times via replication in computing clusters," in *IEEE INFOCOM*, 2015, pp. 1355–1363.

[29] ——, "Evaluating replication for parallel jobs: an efficient approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, pp. 2288 – 2302, 2016.

[30] Z. Qiu, J. F. Pérez, and P. G. Harrison, "Variability-aware request replication for latency curtailment," in *IEEE INFOCOM*, 2016, pp. 1–9.

[31] K. Lee, R. Pedarsani, and K. Ramchandran, "On scheduling redundant requests with cancellation overheads," in *Allerton*, 2015, pp. 99–106.