Reconciling Cost and Performance Objectives for Elastic Web Caches

Farhana Kabir
Washington State University
farhana.kabir@wsu.edu

David Chiu
Washington State University
david.chiu@wsu.edu

***Abstract*—Web and service applications are generally I/O bound and follow a Zipf-like request distribution, ushering in potential for significant latency reduction by caching and reusing results. However, such web caches require manual resource allocation, and when deployed in the cloud, costs may further complicate the provisioning process. We propose a fully autonomous, self-scaling, and cost-aware cloud cache with the objective of accelerating data-intensive applications. Our system, which is distributed over multiple cloud nodes, intelligently provisions resources at runtime based on users cost and performance expectations, while abstracting the various low-level decisions regarding efficient cloud resource management and data placement within the cloud from the user. Our prediction model lends the system the capability to auto-configure the optimal resource requirement to automatically scale itself up (or down) to accommodate demand peaks while staying within certain cost constraints while fulfilling the performance expectations. Our evaluation shows a $5.5\times$ speedup for a typical web workload, while staying under cost constraints.**

I. Introduction

The current generation of computing devices is contributing to a proliferation of data. One of the greatest technological challenges of the 21st century is inarguably how we respond to a new era of computing, which is increasingly being made accessible over the web. A recent foray into meeting this challenge is the advancement of cloud computing. In particular, the cloud's Infrastructure-as-a-Service (IaaS) framework allows for *elastic* computing, *i.e.*, instantaneous pay-as-you-go access to virtually infinite storage and compute resources [1]. Elasticity in this context refers to the ability to allocate capacity on demand and to relinquish that capacity when it is no longer required or when allocation costs reach a certain threshold.

Elasticity has found many uses in capacity expansion for a number of applications [2], [3], [4], [5], [6]. Among these, web and service oriented applications are particular beneficiaries of elastic computing because opportunities abound for intermediate caching and reuse. For instance, consider the ubiquitous three-tier web architecture: Users submit requests to a web interface, which takes the queries and executes a script to retrieve potentially large amounts of data from a database back-end. The retrieved data may be further aggregated and restructured for presentation before returning back to the user. As today's web and service-oriented applications become increasingly more data intensive, the ability to cache precomputed results in the cloud can significantly reduce request latencies [7], [4].

Web traffic can be very dynamic, mostly observing diurnal variability, but can also have unpredictable peaks. In such a variable traffic environment, it would be desirable to have an in-cloud cache that expands and contracts correspondingly to meet performance or SLA requirements. However, managing cloud resources while considering the cost/performance tradeoff is nontrivial. For a cloud based cache to be both cost-effective and efficient, the underlying structure of the virtual storage hierarchy, *i.e.*, machine-memory, local/network disks, and persistent storage, must be considered in terms of costs.

As a classic example, we can consider the growth in April 2008 experienced by *Animoto*, a web service application that produces videos from photos, video clips, and music. The application ramped from 25,000 users to 250,000 users in just 3 short days, signing up 20,000 new users per hour at peak. Animoto went from using 50 back-end servers (Amazon EC2 instances) to over 3,400 in just 3 days and became a cloud computing success story overnight [8]. Undoubtedly, the ability to rapidly scale up on demand is of paramount for a cloud data storage. However, the economics of seemingly limitless capacity to scale up might not be acceptable for all situations. For certain applications, taking a performance hit to keep the cost down might be perfectly reasonable. Our cost and performance models offer a solution focused on resource usage costs to accelerate data-intensive computing.

Our goal is to develop an easily deployable cache on the cloud that autonomously adjusts provisioned IaaS resources based on a user's *cost* and *performance* constraints. By leveraging an artificial neural network for load prediction, we have developed cost and performance models to inform a bi-objective optimization. We will show how models can enable an elastic in-cloud cache to make autonomous decisions on scaling, *i.e.*, expanding/contracting resources in order to gracefully adapt to varying web loads while upholding user preferences on cost and performance without requiring manual intervention.

Our research utilizes the *Amazon Web Services* (AWS) IaaS framework as a testbed. The principal contribution of this research is a dynamic model of an elastic cache, which facilitates the abstraction of various cost/performance tradeoff of cloud resources and graceful cache scaling within a user's

† This research was supported in part by an Amazon Web Services Research Grant and a WSU Seed Research Grant.

budget with the ultimate goal of accelerating web applications. We have evaluated our cloud-based web cache, and we show that we can accelerate web requests for files of the typical size magnitudes by over $5\times$, and even more significantly for data in the tens of MBs of magnitude.

The remainder of this paper is organized as follows. The next section gives an overview of our cache system. Section III describes our models and algorithms. The system evaluation is presented in IV. We present related works in Section V, and conclude in Section VI.

II. System Background

Our cache, situated in the cloud between the web application and users, provides an abstraction to the various nuanced cost-benefit tradeoffs associated with cloud resources. We utilize the Amazon Web Services cloud, which consists of two major services: Elastic Compute Cloud (EC2) and Simple Storage Service (S3). EC2 offers users on demand allocation of virtual machine *instances* at an hourly rate, determined by instance's CPU, memory, and I/O capacity. S3, on the other hand, is a highly reliable persistent store. It allows users to store data objects using an FTP-style interface, and users are charged a rate per GB-month stored. In this section we present the architecture of our elastic web cache, which is depicted in Figure 1.

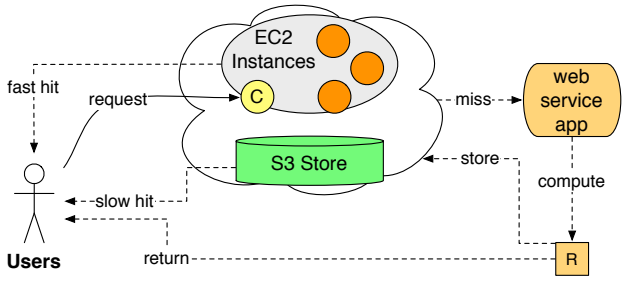


Fig. 1. Elastic Web Cache Overview

The cache is three-tiered: the *Cache Coordinator* (denoted `C` in the figure) receives users' HTTP (SOAP/REST for services) requests and forwards them to the appropriate EC2 instance using consistent hashing [9]. Each EC2 instance stores a portion of the cached data in memory or on disk, which offers comparatively faster I/O relative to S3, but is more expensive in terms of costs. At the time of writing, EC2 pricing ranges anywhere from \$0.08 to \$1.80 per instance-hour allocated, depending on the instance type's capabilities. In this paper, we experimented with a middle-of-the-road `m1.large` instance type, which is \$0.32 per instance-hour for 7.5 GB memory and 4 virtual compute units. Over time, the number of allocated EC2 instances can grow or shrink to handle the current workload.

The data organization on each EC2 instance is a key-value store, using a B^+-Tree [10] to provide fast searches over its key space. Because EC2 can be costly and instance memory is limited, we use the least recently used (LRU) cache replacement scheme [11]. Any evicted data object is migrated down to the S3-level. While S3 is a much cheaper storage option than the EC2-resident cache (roughly \$0.125 per GB-month), its I/O latency is much higher.

Users have the option to input the following two parameters to communicate their preferences: (1) a *cost constraint* C and (2) a *cost-priority* parameter, λ, $0 \leq \lambda \leq 1$. The cost constraint C serves as the upper-bound for the dollar amount that can be spent on the cache per time-unit. This constraint essentially restricts the cache from scaling up uncontrollably in response to a sudden spike in demand and thereby, violating a user's budget. The cost-priority parameter, λ, on the other hand, is a *knob* that allows users to tune the system according to their preference of performance within the limits of their cost constraint. A high value of λ implies that the system should strive to keep costs as far below constraint C as possible. For instance, $\lambda = 1$ should signify our system to configure an S3-only data organization, to save costs, even if C is set to be far greater than the costs associated with the S3-only store. A low value of λ allows the system to aggressively allocate resources to increase performance, while staying just below the budget constraint C.

The *Cache Coordinator* manages the allocation of cloud resources and reconciles the user parameters at all times. For instance, upon any risk of the cache exceeding its current capacity in the near-term, it allocates a new instance, if within user constraints. Conversely, the coordinator may also consolidate instances to reduce cost. To attain the unique features of our cache system (*i.e.*, auto-scaling and cost awareness) the coordinator employs a prediction model that yields a cache configuration with optimal cost and performance for the user.

III. Cost and Performance Optimization

The crux of our self-managed cost aware cache is the cache coordinator's utilization of a mathematical prediction model which employs a *bi-objective optimization* to configure the underlying resource allocation for the cache. The goal of our model is to dynamically analyze cache performance over time and adjust resource requirements in order to strike the appropriate balance to achieve the user's goals in terms of cost and performance expectations.

The two opposing objectives for our system are to *minimize the cost* to store data in the cache and *maximize performance* by allocating enough nodes to facilitate a larger number of hits. Because these two objectives conflict, there will be a cache configuration with the highest performance, another with the lowest cost, and a number of configurations that are compromises between performance and cost. This set of trade-off designs is known as a *pareto set*, and we solve a bi-optimization problem to extract the pareto-optimal solution for our system. To allow for a uniform comparison between performance of various objectives, we normalize both objective measurements to a range between 0 and 1.

Notation	Description
t_F, t_S, t_M	Average latency for a fast hit, slow hit, and miss
H_F, H_S	Fast hit rate, slow hit rate
$EQT(t)$	Effective query time at t
$C_{usage}(t)$	Cache usage cost per hour at t
$C_{min}(t)$	Min possible usage cost per hour at t
$C_{max}(t)$	Max possible usage cost per hour at t
$\hat{Q}(t)$	Predicted number of queries at t
$\hat{L}(t) = \hat{Q}(t) \times D$	Predicted load at future time t
P_{EC2}	Price per EC2 instance-hour
P_{S3}	Price of S3 usage per MB-hour
R_F, R_S, R_M	Data access rate (MBps) on a fast hit, slow hit, and miss, respectively.
T	EC2 node capacity (MB)
D	Average data size (MB)
N	Optimal number of EC2 nodes at t
S	Optimal number of S3 storage used at t

TABLE I
NOTATIONS FOR SYSTEM MODELS

A. Modeling the Performance Objective

From empirical observations, we make the following assumptions in our cache design: (1) data stored in EC2 nodes (either in memory or disk) are retrieved faster than from S3, and (2) the cost per storage-hour is much higher for data stored in EC2 than for S3.

We will first model the performance objective as the *effective query request time* (EQT). If the requested data resides in any of the allocated EC2 instances' memory or disk, it is considered a fast hit (Figure 1) due to faster I/O and data organization. If the requested data is not found in any of the cooperating EC2 instances, we search the persistent store S3, resulting in a slow hit. Clearly, reducing the number of slow hits among the total hits will yield better performance. On a cache miss, the web request or service application is invoked. Its resulting data is sent to the user as well as to the cache. Next, we can define the query request latency as the time between the arrival of a request and retrieval of the queried data.

Equipped with these metrics we formulate the performance objective as follows. For readability, Table I provides a list of the notations used in defining the performance and cost objectives. Because we will observe the number of requests in fixed time intervals ($t = 0, 1, 2, \ldots$), we can model our system discretely. Let

$$Q(t) = Q_F(t) + Q_S(t) + Q_M(t) \tag{1}$$

denote the total number of incoming query requests at time t, where $Q_F(t)$, $Q_S(t)$, and $Q_M(t)$ refer to the number of queries that result in fast hits, slow hits, and misses, respectively. If we further let t_F, t_S, and t_M denote the average query latency for a fast hit, a slow hit, and a miss, then we can define the *effective query request time* at time t as follows,

$$EQT(t) = H_F \times t_F + H_S \times t_S + (1 - H_F - H_S) \times t_M \tag{2}$$

where $H_F = Q_F(t)/Q(t)$ and $H_S = Q_S(t)/Q(t)$ represent the fast hit and slow hit rates respectively. The normalized performance objective is given below,

$$f_p = \frac{EQT(t) - t_F}{t_M - t_F} \tag{3}$$

To inform our algorithms on making resource allocation decisions, we must relate f_p to system parameters that can be adjusted (*e.g.*, number of EC2 nodes that should be allocated). Let $\hat{Q}(t)$ denote the predicted number of requests at time t and let D denote the average data size (MB), then $\hat{L}(t) = \hat{Q}(t) \times D$ is the *predicted* system load in MB at time t. To predict future requests $\hat{Q}(t)$, we use an artificial neural network (ANN) in our implementation [12]. We employ a *back-propagation* algorithm for *feedforward* networks with *sigmoid* activation function to train the ANN. Our reasoning for choosing this particular variant stems from the fact that studies show that this form of ANN offers a simple straight forward extension to a widely used classical way to model time series [13].

We further let T, P_{EC2}, and P_{S3} denote the system parameters from the cloud. Namely, T is an EC2 node's capacity (memory and disk) in MB, P_{EC2} is the price of an EC2 instance per hour, and P_{S3} is the price of S3 usage per MB hour. The goal is to find the optimal number of EC2 nodes, N, and the optimal amount of S3 storage (in MB), S, that should be used by the system at time t. Now we can approximate the optimal values for the above parameters as follows,

$$Q_F \approx N \times T/D \tag{4}$$

$$Q_S \approx S/D \tag{5}$$

$$Q_M \approx (\hat{L}(t) - [N \times T + S])/D \tag{6}$$

where $N \times T$ and S denote the data amount residing in cooperating EC2 nodes, and the data amount residing in S3, respectively. Assuming an LRU replacement policy [11], these approximations are justifiable because the total number of hits are proportional to the number of objects whose values reside in the cache. The same reasoning allows us to approximate the fast hits as the number of objects whose values reside in memory and disk, and the slow hits as the number of objects whose values reside in the persistent storage. The *effective query time*, $EQT(t)$, for the system can be derived as follows,

$$EQT(t) = \frac{1}{\hat{Q}(t)} \left(\frac{N \times T}{R_F} + \frac{S}{R_S} + \frac{\hat{L}(t) - (N \times T) - S}{R_M} \right) \tag{7}$$

where R_F, R_S, and R_M to represent the data access rate (MBps) on a fast hit, a slow hit, and a cache miss, respectively. Also, the lowest and the highest bounds on query latency, *i.e.*,

the average latency on a fast hit and the average latency on a cache miss can be determined as,

$$t_F = \frac{D}{R_F}\ ,\ \ t_M = \frac{D}{R_M} \tag{8}$$

After substitution, we derive our performance objective function,

$$f_p = \frac{\frac{1}{\hat{Q}(t)}\left(\frac{N\times T}{R_F} + \frac{S}{R_S} + \frac{\hat{L}(t)-(N\times T)-S}{R_M}\right) - \frac{D}{R_F}}{\frac{D}{R_M} - \frac{D}{R_F}} \tag{9}$$

B. Modeling the Cost

Depending on a user's cost-performance preferences, the cache coordinator decides where data should be placed in cooperating EC2 instances or in S3. We define the cost objective f_c to be the normalized total usage cost over the various cloud storage options,

$$f_c = \frac{C_{usage}(t) - C_{min}(t)}{C_{max}(t) - C_{min}(t)} \tag{10}$$
$$\text{such that } C_{usage}(t) \geq C_{min}(t)$$

where $C_{usage}(t)$, $C_{min}(t)$, and $C_{max}(t)$ refer to the cache usage cost per hour at time t, the least possible cost per hour (with S3-only configuration), and the maximum possible cost per hour (with an EC2-only configuration), respectively. We note that C_{min} is a lower-bound on C_{usage} , To minimize the cost objective, the goal is to use as few EC2 instances as possible to store the data at time t.

Like before, we must again relate the above cost variables to controllable system parameters.

$$C_{min}(t) = \hat{L}(t) \times P_{S3} \tag{11}$$
$$C_{max}(t) = \hat{L}(t)/T \times P_{EC2} \tag{12}$$
$$C_{usage}(t) = N \times P_{EC2} + S \times P_{S3} \tag{13}$$

where $\hat{L}(t)/T$, $N \times P_{EC2}$, and $S \times P_{S3}$ indicate the number of EC2 nodes required to store all of data in instance memory and disk, the cost for the allocated nodes, and is cost for the persistent storage used, respectively.

After substitution and normalization, the cost objective function can be fully expressed as follows,

$$f_c = \frac{[(N \times P_{EC2}) + (S \times P_{S3})] - (\hat{L}(t) \times P_{S3})}{\left(\frac{\hat{L}(t)}{T} \times P_{EC2}\right) - (\hat{L}(t) \times P_{S3})} \tag{14}$$

C. Solving the Optimization Problem

Recall that there are two user inputs to exploit the cost-performance tradeoff: (1) C the cost constraint per time unit, and (2) λ, $0 \leq \lambda \leq 1$. A higher value of λ implies that the cache coordinator should strive to keep costs as low as possible. Our problem fits classical *weighted sum* approach, which assigns a weight w_i to each normalized objective function $f_i(x)$ so that the problem is converted to an aggregated single-objective problem with a scalar objective function as follows:

$$\underset{x}{\text{argmin}}\, F(x) = w_1 f_1(x) + w_2 f_2(x) + \ldots + w_m f_m(x) \tag{15}$$

where x denotes the system parameters, $f_i(x)$ is the normalized objective function for the i^{th} objective, and $\sum w_i = 1$ for a given weight vector $w = w_1, w_2, \ldots, w_m$. Known also as the apriori approach because it requires the user to provide the weights before optimization can begin, this method yields a single solution. Using the *weighted sum* method, we derive the following scalar objective function for our elastic cache,

$$\underset{x}{\text{argmin}}\, F(x) = (1-\lambda) f_p(x) + \lambda f_c(x) \tag{16}$$

$$\text{subject to: } \forall_t : C_{usage}(t) \leq C$$

which satisfies the user specified cost constraint, C, the maximum allowable usage cost, at all times. Through solving $\text{argmin}_x F(x)$ for the tuple $\langle N, S \rangle$, we obtain the optimal number of nodes and persistent storage to allocate for an application.

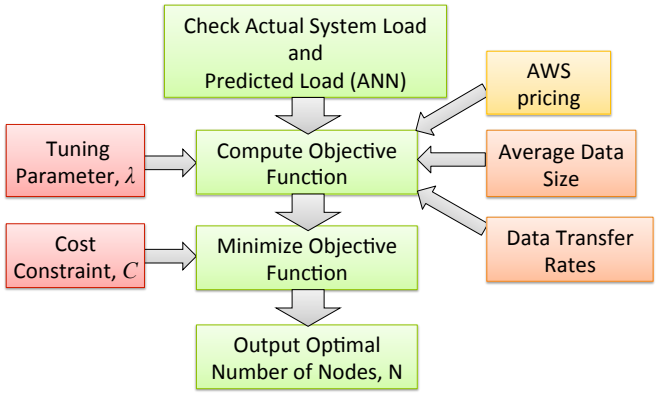


Fig. 2. Steps to Compute Optimal Number of Nodes

To solve our multi-objective optimization problem, we perform a linear search over all possible values of N (number of instances allocated) to find the minimum of the weighted objective function. We argue that realistically this linear search does not increase the time complexity of the algorithm by much since Amazon has constant limits on maximum instance allocation, thereby keeping the value of N small enough to perform an efficient linear search. In the general case, a binary search can be employed to narrow down a likely value for N assuming both objectives are increasing functions.

The major steps in our algorithms towards determining the optimal resource allocation are depicted in Figure 2. Our first step is to narrow the search space for the optimal number of nodes N by only considering the range of nodes that yields $C_{usage}(t) \leq C$, satisfying the user's cost constraint.

Algorithm 1 inputs the user's cost constraint C, the load L, and pricing data for EC2 and S3. This algorithm returns the resource configuration with the highest cost that can be allocated while meeting the user cost constraint C. On lines (1-6), we compute N, the minimum number of EC2 nodes needed to handle a given load L. Lines (4-5) returns a possily smaller number of nodes required to accommodate the data space, giving us an upper bound on performance (and cost). In lines (7-19), we compute the S3 storage allocation that is

required to hold any data objects exceeding the EC2 node storage.

Algorithm 1 maxResources (C, L, P_{EC2}, P_{S3})

1: ▷ *Maximum nodes that can be allocated while staying within budget*
2: $N \leftarrow \lfloor C/P_{EC2} \rfloor$
3: ▷ *If budget too high, only allocate nodes required to accommodate load*
4: **if** $N > \lceil L/T \rceil$ **then**
5: $N \leftarrow \lceil L/T \rceil$
6: **end if**
7: $S \leftarrow 0$
8: ▷ *If load is greater than total node capacity then*
9: ▷ *if budget allows, use $S3$ storage*
10: **if** $L > (N \times T)$ **then**
11: **if** $(C - (N \times P_{EC2})) > 0$ **then**
12: ▷ *Budget allows S MB of $S3$ storage,*
13: $S \leftarrow (C - (N \times P_{EC2}))/P_{S3}$
14: ▷ *but the required storage might be less*
15: **if** $(L - (N \times T)) < S$ **then**
16: $S \leftarrow L - (N \times T)$
17: **end if**
18: **end if**
19: **end if**
20: **return** $\langle N, S \rangle$

While Algorithm 1 optimizes the resource allocation for the given cost constraint, Algorithm 2 further restricts the resource configuration according to the user's cost priority parameter, λ. On line 2, we first retrieve the highest performing $\langle N, S \rangle$ pair under constraint C. Next, we compute the aggregate objective function given in Equation 16 using $\langle N, S \rangle$ and λ. Then on lines (6-13), we iterate over decreasing values of EC2 nodes n_i to recompute the objective function and return the n_i value resulting in the minimum.

Algorithm 2 optimalNodes (C, λ)

1: ▷ *Get the most expensive resource configuration while staying within budget*
2: $\langle N, S \rangle \leftarrow$ maxResources(C)
3: ▷ *Solve optmization (Eq. 16) for the given $\langle N, S \rangle$ pair*
4: $min\_f \leftarrow$ computeObjective(N, S, λ)
5: $opt\_n \leftarrow N$
6: **for** $n_i = N - 1$ downto 0 **do**
7: $s_i \leftarrow (C - (n_i \times P_{EC2}))/P_{S3}$
8: $f_i \leftarrow$ computeObjective(n_i, s_i, λ)
9: **if** $f_i < min\_f$ **then**
10: $min\_f \leftarrow f_i$
11: $opt\_n \leftarrow n_i$
12: **end if**
13: **end for**
14: **return** opt_n

IV. System Evaluation

In this section, we initially describe our experimental setup, followed by evaluation results.

A. Experimental Setup

The environment for our experiments is on Amazon's EC2 cloud. We consider only `m1.large` instances, each having 7.5 GB of memory with 4 virtual EC2 cores where each core has the processing power equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor on a 64-bit platform. Each of these instances is loaded with an Ubuntu Linux image.

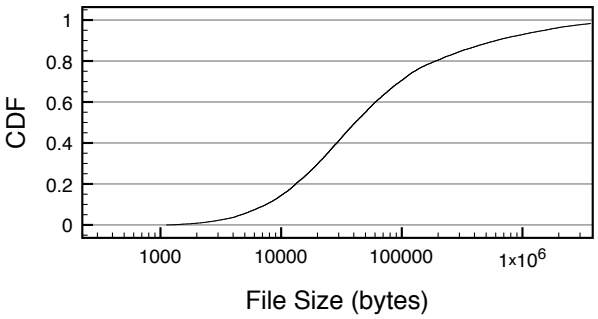
Our web cache is configured as shown in Figure 1. To introduce geographical diversity, we placed our web workload requestor and cache nodes in the `us-east-1` region and the web server in `us-west-2`. The goal for this particular arrangement with a different locality is to introduce a slight network latency. This scenario is quite common for web applications where the front-end servers are distributed around the world at the *edge* of the web to place them in proximity to the users, and yet rely on a distant back-end server. We used the Surge web traffic generator [14] to produce 15,000 file objects (amounting to 4GB) and 200,000 HTTP GET requests on these objects over a Zipf distribution. These files are stored and served by the `us-west-2` node. Our cache is essentially an in-core *key-value* store, where *key* represents the filename and *value* being the file content.

The *cache coordinator*, deployed on a large EC2 instance, listens for and responds to client requests. The coordinator is also responsible for monitoring the overall cache capacity and subsequently allocating/deallocating additional cooperating EC2 server nodes within the same AWS region that work together to provide the optimal elastic cache storage for the user.

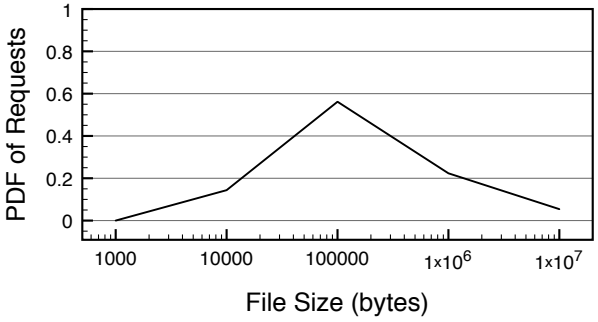
We will now discuss the experimental setup behavior through the statistical distribution of the data set and the traffic requests. Figure 3(a) demonstrates the cumulative distribution function (CDF) of file sizes for the 15,000 files in the data repository. The files in our experimental setup range from 1KB to 10MB in size, with over 50% of those files falling between 10KB to 100KB range, as seen in the CDF. To show the workload distribution, Figure 3(b) depicts the probability density function (PDF) of a request for a particular file of a certain magnitude in our data set over the 200,000 HTTP requests that are generated.

B. Cache System Evaluation

The empirical validation of our original hypothesis of users availing our self-managed cache to accelerate Web services applications makes up this section. Figure 4 juxtaposes the average query latencies ensuing from two separate experimental runs of our system, one utilizing the cache and the other bypassing it altogether. The horizontal axis of the graph shows the total number of HTTP requests processed as time advances. The vertical axis exhibits the average request latency, which is averaged every for 1000 requests processed.



(a) CDF of File Sizes on Server



(b) PDF of Request Size

Fig. 3. File and Request Distributions

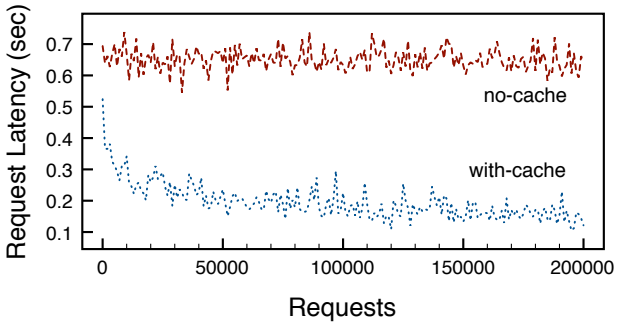


Fig. 4. Query Request Latency

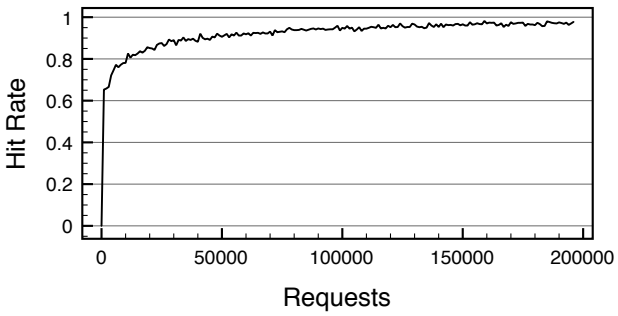


Fig. 5. Hit Rate

A significant speedup is evident after only a few thousand requests, due to the Zipf-based workload distribution [15], which web requests generally follow. It is apparent from the trend of the graph in Figure 4 that the files with high request probability make their way into the cache towards the beginning of the long experimental run, thereby, generating fast hits for subsequent requests and commencing a fast converging trend for the average query latency. At the end of this run, we observed a $5.52\times$ average speedup per request. The request hit-rate trend depicted in Figure 5 further corroborates these findings, as we approach a hit-rate of 100%. Again, we can observe that most hits occur early due to the skewed request distribution.

We further analyze the average latency trajectory displayed by our system in Figure 6. In this graph, we focus only on the first 3,500 requests (where the most interesting behavior can be seen). We have also disaggregated the latency to varying exhibited per file magnitude range. As seen previously in Figure 3(a), our entire data set of 15,000 files can be disaggregated into the following four file size magnitudes: `10K`, `100K`, `1M`, and `10M`. For instance, the `100K` data set consists of all files $\geq$ 100KB and $<$ 1MB. Figure 6 shows

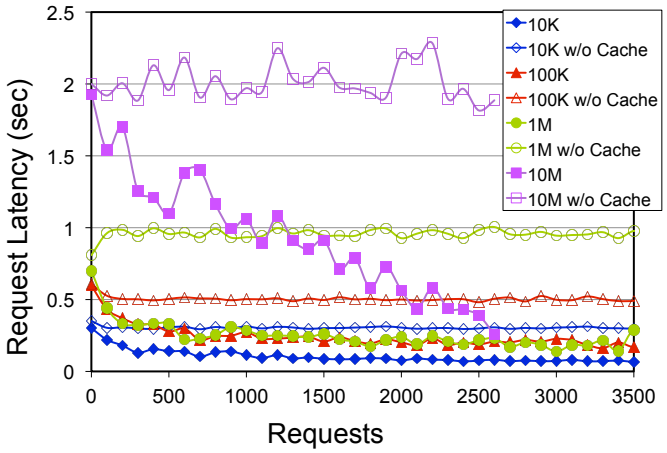


Fig. 6. Average Latency Drop per File Magnitude

the query latencies observed per disaggregated data set, both with and without employing the cache. Expectedly, the latency drop is highest (close to $10\times$ speedup) for the containing files ranging in magnitude from 1MB to 10MB. The charts per data set in Figure 6 confirms that our cloud-based web cache can accelerate applications requiring large data movement for data-intensive applications. Next, we evaluate the cost and performance models.

C. Cost-Performance Model Evaluation

The mathematical model presented in Section III calibrates the overall size of our cache based on user input on cost constraint C, and the cost-performance tuning parameter λ. For the performance evaluation presented in the previous subsection, we deliberately set $\lambda = 0$ to indicate user preference for the highest performing system. We also did not place any limiting budget constraint in the previous experiment as the

primary focus of the experiment was to evaluate the fitness of our system in terms of speed. Hence the optimal cache configuration, determined by the model, accommodated the entire load in EC2 node(s). This section is a departure from the performance-only ethos as we demonstrate the mathematical model's behavior when faced with limiting cost constraints and user preferences.

In this experiment, to garner observable results in a reasonable amount of time, we stipulate the average data size to be 50MB. Furthermore, we restrict the capacity of a single node to 500GB allowing our resource allocation algorithm to indicate a need for scaling up in a short period time. Figure 7 illustrates the cost-performance model's predilection towards scaling up amid a constant request rate of 25 per second. The left vertical axis shows the EC2 nodes allocated, while the right vertical axis shows the cost per hour incurred. The cost constraint $C = \$0.75$/hr is shown as a bold horizontal line. We show the results for $\lambda = 0$, 0.25, and 0.5, where λ close to 0 denotes a user's desire for higher performance. Note that in all three cases, we are able to stay under C, while a lower λ yields slightly more nodes (higher performance by caching more files).

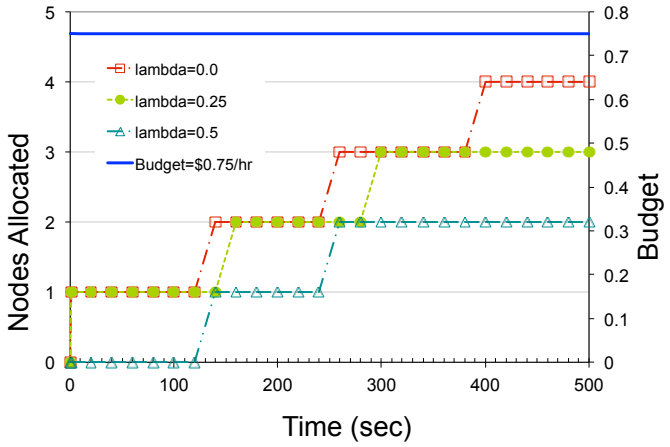


Fig. 7. Optimal Node Allocation (Request Rate: 25 requests/sec)

Figure 8 highlights the results of a set of experiment with a constant query rate of 100 requests per second. It is evident from the graph that the system is decidedly slow to scale up by allocating a new EC2 node for larger λ values, *i.e.*, placing more emphasis on savings than performance, commissioning the overflow data to the slower persistent storage. Note that the user's budget is set to $0.75/hour and allocating 5 nodes would exceed that budget requiring a price tag of $0.8 per hour, our model never indicates the need for more than 4 nodes, even in the highest performing mode, as that would violate the cost constraint. The optimal number of nodes is configured to 4 even though it forces a certain fraction of the data to be placed in Amazon's S3 or be evicted out of the cache as the load increases with time. Furthermore, we observe that the model's decision to allocate a new node is deferred until later with a significant increase in the size of the load as λ value increases. For λ values 0.25 and 0.5, the highest number of

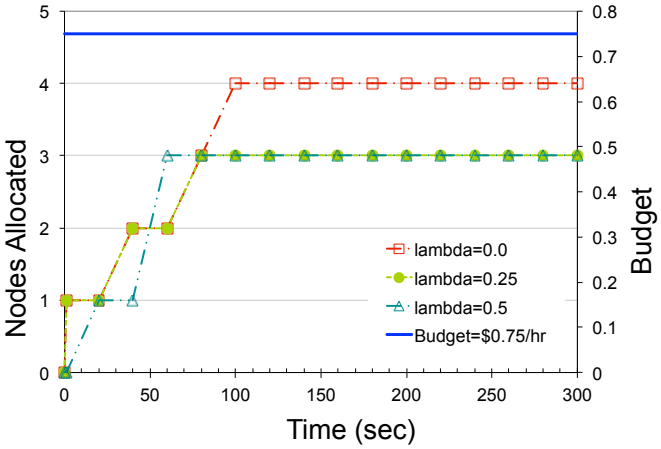


Fig. 8. Optimal Node Allocation (Request Rate: 100 requests/sec)

nodes the system will allocate is 3 to achieve the optimal balance between cost and performance.

These results demonstrate that our elastic cache can successfully reconcile cost and performance objectives. Furthermore, we show that our cache can be easily scaled to accelerate data-intensive web applications.

V. Related Works

This section summarizes the related state-of-the-art research in the area of cloud resource allocation in order to achieve automated scaling.

Although most cloud providers offer only a cloud management API and expect users to implement their own software stack to manage their compute resources, AWS AutoScaling automates resource provisioning to some degree based on user defined policies on infrastructure level performance metrics [16]. Essentially, users specify threshold values for performance, and whenever the observed performance metric goes above or below the threshold, a predefined number of compute nodes are added or removed from the application's resource pool. These simple mechanisms work well for easily parallelizable applications, *e.g.*, Map-Reduce applications [17]. However, in cases where much more distributed coordination is required, these mechanisms render themselves inadequate and elasticity does not directly translate to scalability [4]. In contrast, our system implements a more fine grained scaling logic. Additionally, our system takes into account a user's budget when making scaling decisions.

Mao, *et al.* presented a dynamic cloud scaling mechanism which can automatically scale up or scale down the underlying cloud infrastructure based on job deadlines [18], [19]. The authors posit that an infrastructure based metric is not reflective of the quality of service (QoS) a cloud application is providing or user's performance expectations. In contrast, our primary focus is to never violate the budget constraint put in place by the user. Shen, *et al.* describe *CloudScale*, a system to reduce prediction errors in a prediction-driven elastic resource scaling for multi-tenant cloud computing infrastructures [20]. The error correction method minimizes the impact of resource

under-estimation errors to minimize SLA violations with low resource waste.

Closer to our work, Zhu, *et al.* make a case for scaling down the caching tier of multi-tiered cloud based web services for potentially huge cost savings while maintaining a viable performance to meet the SLA [21]. The authors posit that although scaling down the caching tier increases cache misses, with an overall drop in the load, an application can afford to let more requests into the data tier without SLA violation. To correctly size the caching tier they propose working backwards, *i.e.*, to determine the minimum cache hit-rate needed to ensure a response time meeting the SLA, and then calculating the cache size that would provide that hit-rate.

Amazon recently announced their in-memory caching service in the cloud, *ElastiCache* [22]. The service aims at improving the performance of web applications by allowing them to retrieve information from the fast in-memory cache as opposed to slower disk-based databases. Although the users have the option to scale ElastiCache to tailor it to their requirements, it does not scale automatically without explicit user intervention. *Memcached* [7], [23], a widely adopted memory object caching system, is a distributed in-memory cache for small chunks, up to 1 MB in size, of arbitrary data resulting from database calls, API calls, web page rendering *etc.* The system is designed to speed up dynamic web applications by alleviating database load. Its simple design enforces an LRU eviction policy upon reaching capacity.

Our system is fundamentally unique from all of these available data caching technologies. Our cache can utilize both memory or disk storage and has the capability to auto configure the optimal resource requirement based on user's preference on cost and performance. It also has the capacity to automatically scale itself up/down to gracefully accommodate demand surge/lulls.

VI. Conclusion and Future Work

While researching load prediction mechanisms befitting our self-scaling cache, we observed that predicting network traffic is of significant interest in the network management domain in order to implement policies regarding congestion control, admission control, *etc.* Among the multitude of approaches that exist for network load prediction, the algorithms that utilize time series, much like our work, seem worth exploring. We identify a small body of work that looks promising in the context of our cache. A prediction algorithm by Zhao *et al.* shows promise for predicting short term bursty traffic [24]. The proposed algorithm takes advantage of multiple time scales in time series to extract statistical properties of network traffic as opposed to a single one.

In this paper, we proposed utilizing the IaaS cloud computing paradigm for the purposes of caching web and service applications while staying within any cost constraints imposed by the user. We modeled the cost-performance tradeoff for the resource allocation of such a cache as a bi-objective optimization problem. Our system evaluation shows that our resource allocation algorithm allows users to effectively tune performance requirements while staying within budget.

References

[1] M. Armbrust, *et al.*, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[2] M. D. de Assuncao, A. di Costanzo, and R. Buyya, "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," in *Proceedings of HPDC'09*. ACM, 2009, pp. 141–150.

[3] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *Proceedings of CCGrid'10*, 2010, pp. 43–52.

[4] D. Chiu, A. Shetty, and G. Agrawal, "Elastic cloud caches for accelerating service-oriented computations," in *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'10), New Orleans, LA, USA*, November 2010, pp. 1–11.

[5] M. Cardosa, *et al.*, "Exploring mapreduce efficiency with highly-distributed data," in *MapReduce'11*. ACM, 2011, pp. 27–34.

[6] T. Bicer, D. Chiu, and G. Agrawal, "Time and cost sensitive data-intensive computing on hybrid clouds," in *Proceedings of the 2012 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, 2012.

[7] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, August 2004.

[8] "The Rightscale Blog," http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/.

[9] D. Karger, *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, 1997, pp. 654–663.

[10] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA: ACM, 1970, pp. 107–141.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proceedings of SIGMOD'93*. New York, NY, USA: ACM, 1993, pp. 297–306.

[12] "Fast Artificial Neural Network," http://leenissen.dk/fann/wp/.

[13] R. Frank, N. Davey, and S. Hunt, "Time series prediction and neural networks," *Journal of Intelligent and Robotic Systems*, vol. 31, no. 1, pp. 91–103, 2001.

[14] P. Barford and M. E. Crovella, "Generating representative Web workloads for network and server performance evaluation," in *Proceedings of Performance '98/SIGMETRICS '98*, Jul. 1998, pp. 151–160. [Online]. Available: http://www.cs.bu.edu/faculty/crovella/paper-archive/sigm98-surge.ps

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *Proceedings of Infocom*, 1999.

[16] "Amazon Auto Scaling," http://aws.amazon.com/autoscaling/.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[18] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proceedings of 11th ACM/IEEE International Conference on Grid Computing, GRID 2010, Brussels, Belgium*, October 2010.

[19] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC11, Seattle, WA, USA*, November 2011.

[20] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC'11, Cascais, Portugal*, October 2011.

[21] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. Kozuch, "Saving Cash by Using Less Cache," in *HotCloud '12*, Boston, MA, June 2012.

[22] "Amazon ElastiCache," http://aws.amazon.com/elasticache/.

[23] "Memcached," http://memcached.org/.

[24] W. Zhao and H. Schulzrinne, "Predicting the Upper Bound of Web Traffic Volume Using a Multiple Time Scale Approach," in *Proceedings of WWW'03*, Budapest Hungary, 2003.