Elastic Resource Allocation for a Cloud-Based Web Caching System

FARHANA KABIR, TRAVIS HALL, SCOTT A WALLACE, DAVID CHIU
Washington State University

Web and service applications are generally I/O bound and follow a highly skewed 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 user's cost and performance expectations while abstracting away the various low-level decisions regarding efficient cloud resource management and data placement within the cloud. 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 and fulfilling the performance expectations.

Keywords: cloud, data management

1. INTRODUCTION

One of the great technological challenges of the 21st century is inarguably how we respond to a new era of computing, which is increasingly data-intensive and web-based. 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 [Armbrust*, et al.* 2009]. 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 right-sizing for many cloud-based applications [de Assuncao et al. 2009; Das et al. 2009; Lin et al. 2010; Marshall et al. 2010; Cardosa*, et al.* 2011; Bicer et al. 2012]. Among these, web and service oriented applications are particular beneficiaries, since opportunities abound for intermediate caching and reuse. For instance, consider the ubiquitous three-tier web architecture: Users submit requests to an HTTP interface, which takes the queries and executes a script to retrieve or compute potentially large amounts of data from a database backend. The retrieved data may undergo further processing, aggregation, and restructuring before it is returned back to the user. As today's web and service-oriented applications become increasingly more data-intensive, caching likewise becomes more important since precomputed web data can significantly reduce request latencies [Karger*, et al.* 1999; Fitzpatrick 2004; Chiu et al. 2010].

Web traffic can be very dynamic and observe unpredictable spikes and troughs. In such a variable traffic environment, it would be desirable for an organization to have an in-cloud cache that expands correspondingly to meet performance or Service-Level Agreements (SLA) requirements, while contracting resources to keep costs down. 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 three short days, acquiring 20,000 new users per hour at its peak. Animoto went from using 50 back-end servers (Amazon EC2 instances) to over 3,400 over this period to become a cloud computing success story overnight [Eicken 2008]. Undoubtedly, the ability to rapidly scale up on-demand is paramount for a storage provider. However, the economics of seemingly limitless

capacity to scale up might not be acceptable for all situations.

Unfortunately, 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. For certain applications, taking a hit on performance 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. In this paper, we present a system that performs this function. We show how cost and performance 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, that facilitates various cost/performance tradeoff abstractions with the ultimate goal of accelerating web applications while remaining within the constraints of a user's budget. We have evaluated our cloud-based web cache where we demonstrate a 5× acceleration for files of typical size magnitudes and even larger acceleration for data files with size magnitudes in the tens of MBs.

The remainder of this paper is organized as follows. Section 2 presents an overview of our cloud-based cache system. Section 3 describes our models and algorithms. The system evaluation is presented in 4. We present related work in auto-scaling, web caching, and web replacement policies in Section 5. Finally, we discuss future work and conclude in Section 6.

2. SYSTEM OVERVIEW

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.

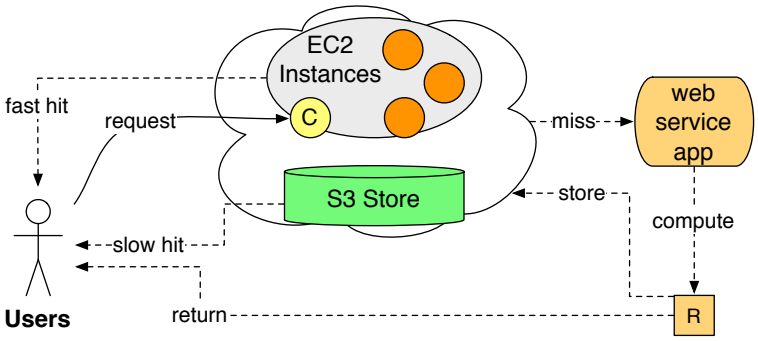


Figure 1. Elastic Web Cache Overview

The cache is three-tiered: the *Cache Coordinator* (denoted `C` in the figure) receives users' HTTP (SOAP/REST for service) requests and forwards them to the appropriate EC2 instance. Each EC2 instance stores a portion of the cached data. As the cache fills over time, instance

data can be replaced, which involves an eviction down to the S3 level, or promotion from S3. To reconcile costs, the number of allocated EC2 instances can grow or shrink to handle the current workload. 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 an `m1.large` instance type, which is $0.32 per instance-hour for 7.5 GB memory and 4 virtual compute units.

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, $\lambda,\ 0 \leqslant \lambda \leqslant 1$. The cost constraint C serves as the upper-bound for the dollar amount that can be spent on the cache per unit time. 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.

2.1 Global Data Organization

At the heart of the storage system lies EC2 nodes, which act as data servers, storing data in instance memory and on disk. We also utilize the considerably more economical persistent storage (S3) options if user's cost preference prohibits hosting all of the data in the more expensive EC2 nodes. Along with the basic `search`, `insert`, and `retrieve` functionalities, each of the data nodes is also equipped with the capability to `migrate` and `evict` data out of the node if necessary. Upon exceeding capacity, a node might need to migrate data to another instance or S3.

The cache coordinator needs to store a global view of all data objects. To this end, it needs a cogent mechanism to index the data servers. Since we are considering our cache under an elastic environment, nodes may scale on demand, and adding or removing cache storage nodes should take minimal effort. This dynamism renders many hashing mechanisms useless as an incredibly large number of key-value pairs would require a rehash to reflect their node membership changes. To address this problem of hash disruption, we employ *consistent hashing* [Karger*, et al.* 1997; 1999] across the EC2 nodes. Consistent hashing lends itself gracefully to any system requiring quick adaptation due to nodes frequently joining and leaving a cooperating system and is used extensively in highly volatile Peer-to-Peer systems, among others.

In consistent hashing, a function $h(k)$ hashes a given key k into a range $[0, n)$. This key range is organized in a clockwise fashion, such that 0 is the successor to $n-1$. At any time, there maybe $m \leqslant n$ EC2 instances (known as buckets) attached to a corresponding hash value. A key k is found in this system by first applying $h(k)$, then identifying the EC2 instance as the first bucket attached moving in a clockwise fashion from $h(k)$. The EC2 instance is finally searched locally for k. For instance, consider Figure 2 and suppose we have three EC2 instances storing data, and they are attached to b_0, b_1, and b_2 on the hash clock. After hashing a key k to the range between b_2 and b_0, we move clockwise to $h(k)$'s successor b_0 and then look for the data object identified with k.

To understand how consistent hashing reduces hash disruption, let us assume that the key range between b_i and b_j is too large, causing the EC2 node attached to b_j to be overloaded

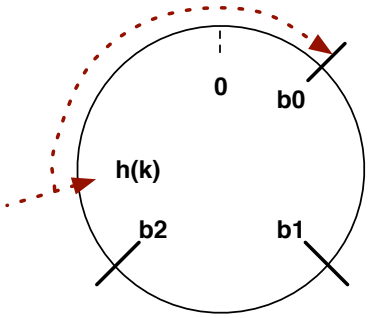


Figure 2. Consistent Hashing Example

with data objects and traffic. To reduce traffic, we could allocate a new EC2 node at b_{split} halfway between b_i and b_j. Then b_j would only have to migrate the keys falling within the range $[b_{i+1}, b_{split}]$ to b_{split}. With traditional hashing algorithms, such an action may require a complete rehash of *all* keys in the system. In general, to insert a new EC2 node into this cache system, the newly allocated node can be placed at b_{split},

$$b_{split} = \begin{cases} \lfloor \frac{b_i + b_j}{2} \rfloor, & \text{if } b_i < b_j \\ \left(b_i + \lfloor \frac{n - b_i + b_j}{2} \rfloor \right) \mod n, & \text{otherwise} \end{cases} \quad (1)$$

Conversely, when an EC2 node located at b_i is deallocated from the system, then it only needs to migrate all of its data objects to its clockwise successor, b_j. It should be noted that this method of locating b_{split} is for illustration only. Realistically, we would want to transfer half the keys from from b_j to b_{split}, which would be a function on the key distribution, not the key range.

Once a data node has been identified, it must be further searched for the cached data results. The next subsection describes the data organization on each local node.

2.2 Local Data Organization

The data organization *within* each EC2 instance uses a B$^+$-Tree [Bayer and McCreight 1970] to provide fast searches over its private key space. Figure 3 shows a simple B$^+$-Tree with cardinality (or branching factor) $p = 2$. Each tree node then contains 2 values and $p + 1 = 3$ pointers down to the next level of the tree. A search for key k starts from the root node, finds the closest value d to k. Then similar to a binary tree, if $k > d$, we traverse down the right edge, and the left edge otherwise. This traversal continues recursively until we reach a leaf node. The leaf node is searched for k, and if it exists, returns the data object pointed by k. It is worth noting that the leaf nodes are linked together and are sorted by key, offering fast response to range queries. Because EC2 can be costly and instance memory is limited, we begin replacing data objects after a certain threshold is reached using the popular least recently used (LRU) policy [O'Neil et al. 1993] . An evicted data object is transferred 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.

3. COST AND PERFORMANCE OPTIMIZATION

The crux of our cloud-based cache is the coordinator's performance and cost model, which informs a *bi-objective optimization* to in determining cloud resource allocation. The goal of our system is to predict 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

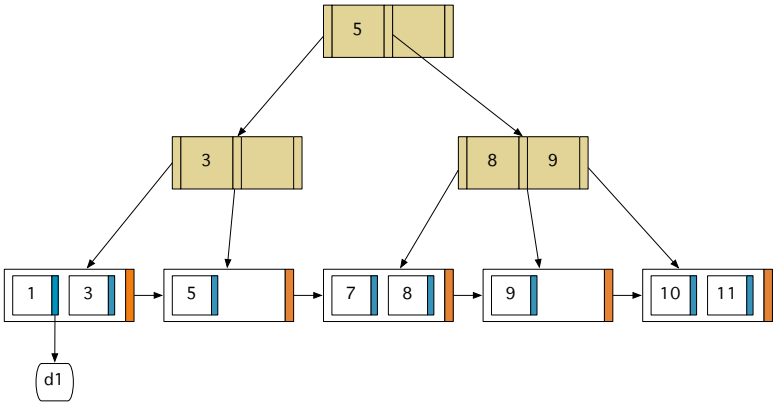


Figure 3. A B$^+$-Tree Example

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

3.1 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

$$Q(t) = Q_F(t) + Q_S(t) + Q_M(t) \tag{2}$$

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,

$$EQT(t) = H_F \times t_F + H_S \times t_S + (1 - H_F - H_S) \times t_M \tag{3}$$

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,

$$f_p = \frac{EQT(t) - t_F}{t_M - t_F} \tag{4}$$

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 request $\hat{Q}(t)$, our implementation uses a feedforward multi-layer artificial neural network (ANN) [Nissen 2003]. The ANN is trained on historic queries using backpropagation with a sigmoid activation function. The multi-layer topology allows the neural network to capture non-linear relationships in the time series data that may be both more subtle and more complex than the relationship captured by, for example, a simple moving average.

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,

$$Q_F \approx N \times T/D \tag{5}$$

$$Q_S \approx S/D \tag{6}$$

$$Q_M \approx (\hat{L}(t) - [N \times T + S])/D \tag{7}$$

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 [O'Neil et al. 1993], 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,

$$EQT(t) = \frac{1}{\hat{Q}(t)} \left(\frac{N \times T}{R_F} + \frac{S}{R_S} + \frac{\hat{L}(t) - (N \times T) - S}{R_M} \right) \tag{8}$$

where R_F, R_S, and R_M 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,

$$t_F = \frac{D}{R_F} \ , \ \ t_M = \frac{D}{R_M} \tag{9}$$

After substitution, we derive our performance objective function,

$$f_p = \frac{\frac{1}{\hat{Q}(t)} \left(\frac{N \times T}{R_F} + \frac{S}{R_S} + \frac{\hat{L}(t) - (N \times T) - S}{R_M} \right) - \frac{D}{R_F}}{\frac{D}{R_M} - \frac{D}{R_F}} \tag{10}$$

3.2 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,

$$f_c = \frac{C_{usage}(t) - C_{min}(t)}{C_{max}(t) - C_{min}(t)} \tag{11}$$
$$\text{such that } C_{usage}(t) \geq C_{min}(t)$$

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.

$$C_{min}(t) = \hat{L}(t) \times P_{S3} \tag{12}$$

$$C_{max}(t) = \hat{L}(t)/T \times P_{EC2} \tag{13}$$

$$C_{usage}(t) = N \times P_{EC2} + S \times P_{S3} \tag{14}$$

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,

$$f_c = \frac{[(N \times P_{EC2}) + (S \times P_{S3})] - (\hat{L}(t) \times P_{S3})}{\left(\frac{\hat{L}(t)}{T} \times P_{EC2} \right) - (\hat{L}(t) \times P_{S3})} \tag{15}$$

3.3 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, $0 \leqslant \lambda \leqslant 1$. 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:

$$\underset{x}{\operatorname{argmin}} F(x) = w_1 f_1(x) + w_2 f_2(x) + \ldots + w_m f_m(x) \tag{16}$$

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,

$$\underset{x}{\operatorname{argmin}} F(x) = (1-\lambda) f_p(x) + \lambda f_c(x) \tag{17}$$

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

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

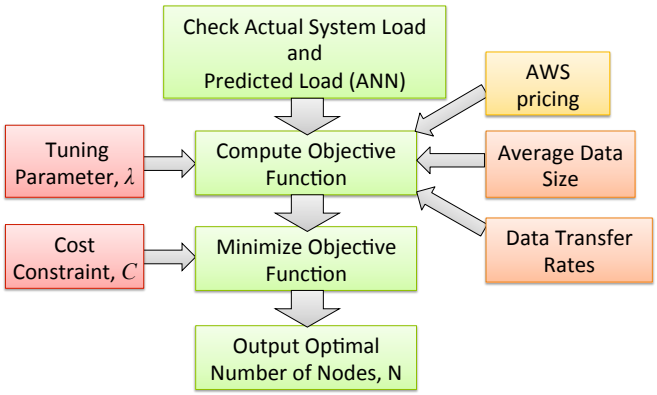


Figure 4. Steps to Compute Optimal Number of Nodes

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 4. 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 $C_{usage}(t) \leqslant C$, 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 possibly 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 ← ⌊C/P_EC2⌋
3: ▷ If budget too high, only allocate nodes required to accommodate load
4: if N > ⌈L/T⌉ then
5:    N ← ⌈L/T⌉
6: end if
7: S ← 0
8: ▷ If load is greater than total node capacity then
9: ▷ if budget allows, use S3 storage
10: if L > (N × T) then
11:    if (C − (N × P_EC2)) > 0 then
12:       ▷ Budget allows S MB of S3 storage,
13:       S ← (C − (N × P_EC2))/P_S3
14:       ▷ but the required storage might be less
15:       if (L − (N × T)) < S then
16:          S ← L − (N × T)
17:       end if
18:    end if
19: end if
20: return ⟨N, S⟩
```

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 17 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: ⟨N, S⟩ ← maxResources(C)
3: ▷ Solve optimization (Eq. 17) for the given ⟨N, S⟩ pair
4: min_f ← computeObjective(N, S, λ)
5: opt_n ← N
6: for n_i = N − 1 downto 0 do
7:    s_i ← (C − (n_i × P_EC2))/P_S3
8:    f_i ← computeObjective(n_i, s_i, λ)
9:    if f_i < min_f then
10:      min_f ← f_i
11:      opt_n ← n_i
12:   end if
13: end for
14: return opt_n
```

4. SYSTEM EVALUATION

We deployed out system entirely over the Amazon Elastic Compute Cloud (EC2) using the experimental configuration shown in Figure 5. We placed the workload generator and all cache nodes in a data center belonging to the same geographical region `us-east-1` while the data

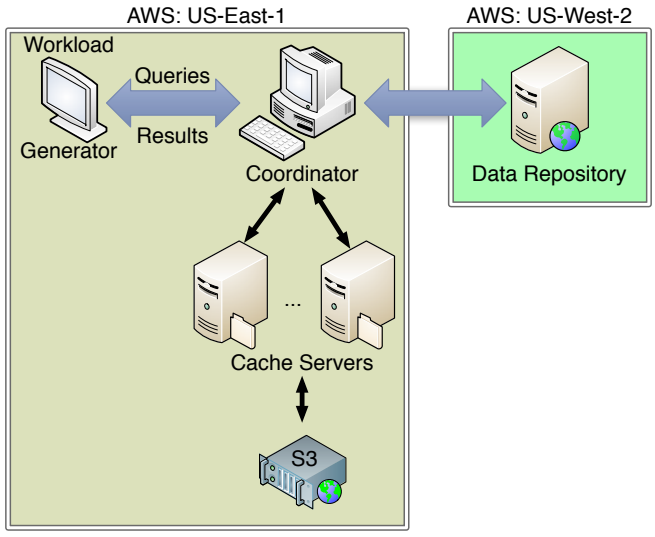


Figure 5. Experiment Layout

repository was located across the continent in `us-west-1`. This configuration was designed to impose a heavy miss penalty.

4.1 Effects of S3 Integration

Each cache server (data node) is running on a 64-bit Ubuntu 10.10 EC2 image. The B^+-Tree is implemented using Ruby 1.9.2p180 (MRI). Each image is launched on an *Extra Large instance* (`m1.xlarge`), which, by Amazon's specifications contains 15GB of memory, 8 EC2 compute units (each compute unit is equivalent to a 1.0-1.2GHz 2007 Xeon or 2007 Opteron processor) and has "high" I/O performance.

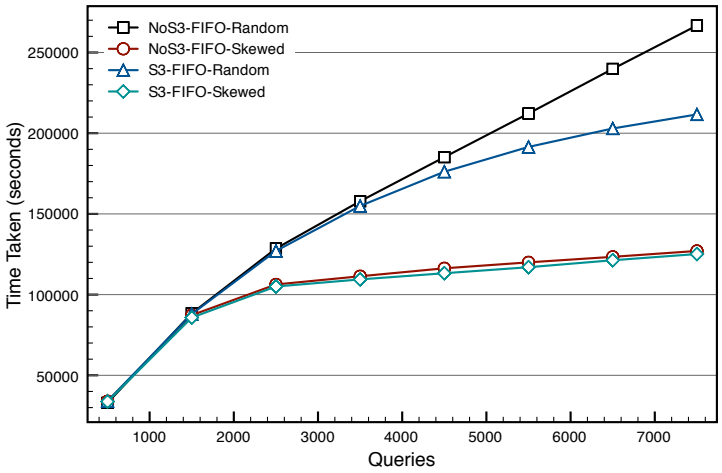
One experimental data setting (the first row of Table II) is designed such that there are a large number of queries over a fairly large number of keys. However, each data object is small enough (5MB) that the cache can maintain a significant portion of them (roughly 2/3) in memory. In this setting, the data node will start replacing after the threshold of 2,000 objects have been reached. A total of 8000 requests will be made under this setting. Another setting uses a larger data size (25MB), which limits the number of objects maintained in memory by the cache. We have 500 objects under this setting, and a total of 1,000 requests are made.

Table II. Experimental Data Settings

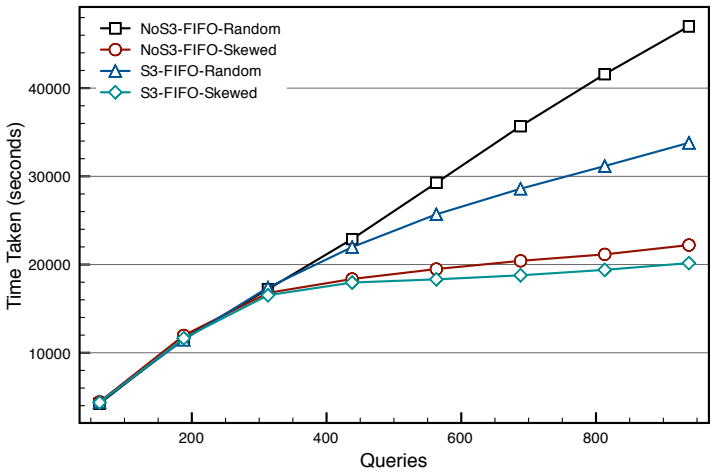
Data Size	Threshold	Objects	Requests
`5MB`	2000	3000	8000
`25MB`	200	500	1000

Further, we change the way our experiment is configured based on three parameters: *Eviction Storage*, *Eviction Strategy*, and *Query Distribution*. *Eviction Storage* has two options: `S3`, meaning that data is evicted from the cache and into S3; and `None`, meaning that data is removed entirely. *Replacement Policy* varies between `FIFO` (First-In-First-Out) and `LRU` (Least-Recently Used). Finally, *Query Distribution* varies between `Random` and `Skewed`. The `Random` query distribution means that the data objects requested are generated by pseudorandom generator. We utilize Ruby's *Kernel::rand* function for this purpose. Ruby 1.9.2 uses a modified Mersenne

Twister with a period of $2^{19937} - 1$ [Ruby Documents]. In contrast, the `Skewed` query distribution means that the requested keys are distributed such that 90% of the queries fall within 10% of the key range.



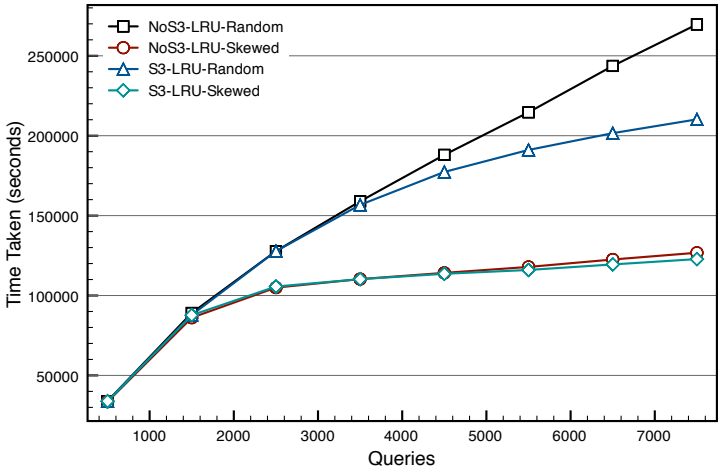
(a) 5MB data files



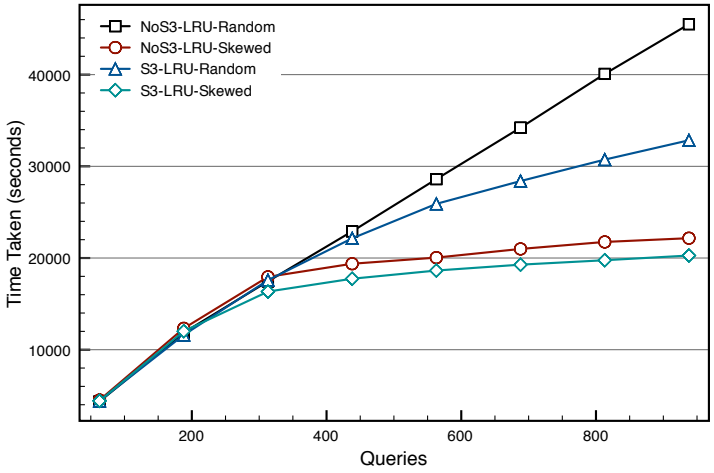
(b) 25MB data files

Figure 6. Time taken using FIFO

In Figure 6, we show the results using the `FIFO` replacement scheme. As expected, the skewed distributions execute in significantly less time regardless of their storage options. This is due, in both cases, to the fact that 10% (the skewed portion upon which 90% of queries fall) of the key range is well within the bounds of the threshold (300 and 50 keys accordingly). As an additional result, we do not see `S3` providing quite so large a benefit as with the random distribution—with the small set of keys retained in-memory for the last half of queries, its benefit only plays out for the first half—shaving off approximately 2,000 seconds (33 minutes) in both the case of the `5MB` (1.5%) and `25MB` (10%). That said, the benefit for randomly distributed queries is significant. With `5MB` data files, we observe a difference of 55,000 seconds (15.3 hours), a 21% reduction in



(a) 5MB data files



(b) 25MB data files

Figure 7. Time taken using LRU

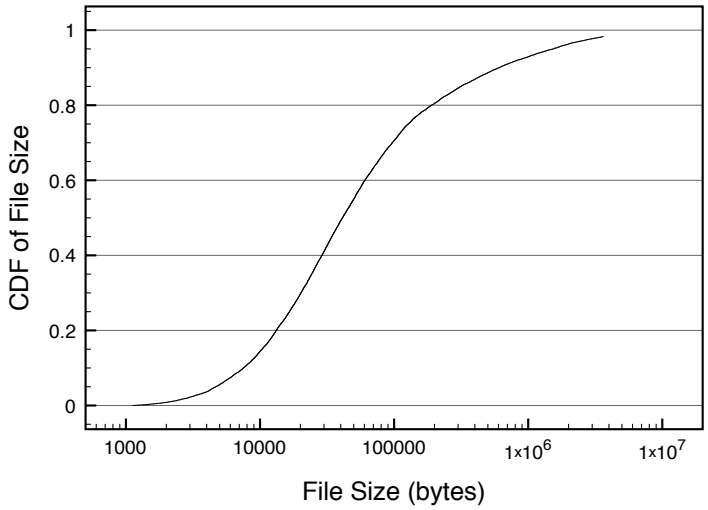
time taken. For `25MB` data files, we can save 13,200 seconds (3.7 hours), or approximately 28% reduction.

In Figure 7, we make notice of the same patterns using the `LRU` eviction strategy. With the `5MB` files, `S3` saves approximately 60,000 seconds (16.5 hours or 22%) for the random distribution and 4,000 seconds (1.1 hours or 3%) for the skewed distribution. Finally, with `25MB` files, we see a reduction of 12,600 seconds (3.5 hours or 28%) and 1,900 seconds (32 minutes or 8%) for random and skewed distributions respectively. These results show that given `FIFO` or `LRU` replacement schemes, evicting data objects to S3 does not offer significant speedup for skewed queries. However, in applications where query distributions are random, S3 provides a substantial speedup.

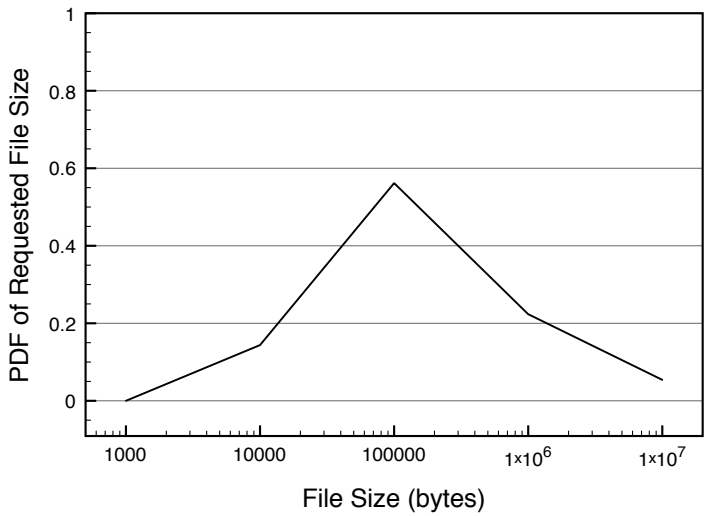
4.2 Elastic Caching Evaluation

The next set of experiments evaluates our models and optimization heuristic to obtain elastic resource allocation. We used the Surge web traffic generator [Barford and Crovella 1998] to

produce 15,000 file objects (amounting to 4GB). Figure 8(a) shows the cumulative distribution function (CDF) over the file sizes that were generated for the experiments. Surge also generated 200,000 HTTP GET requests on these file objects. Figure 8(b) shows the probability density function (PDF) of the file sizes being requested.



(a) CDF of File Sizes on Server



(b) PDF of Request Size

Figure 8. File and Request Distributions

Figure 9 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 significant speedup is evident after only a few thousand requests, due to the Zipf-based workload distribution [Breslau et al. 1999], which web requests generally follow. It is apparent from the trend of the graph in Figure 9 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

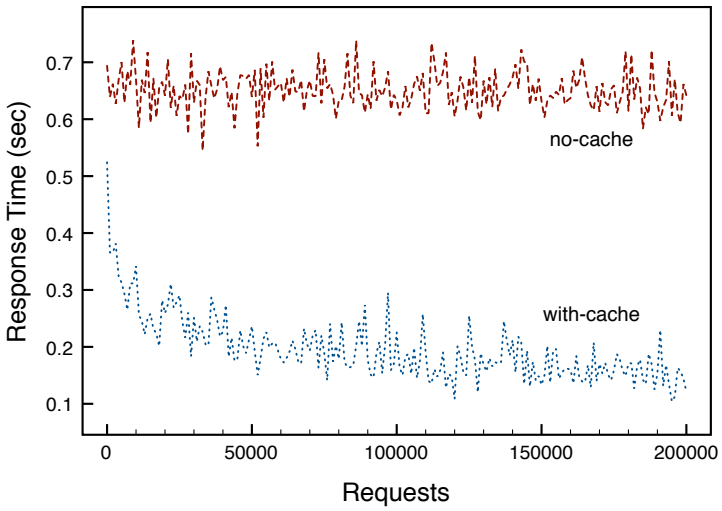


Figure 9. Query Request Latency

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× average speedup per request. The request hit-rate trend depicted in Figure 10 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.

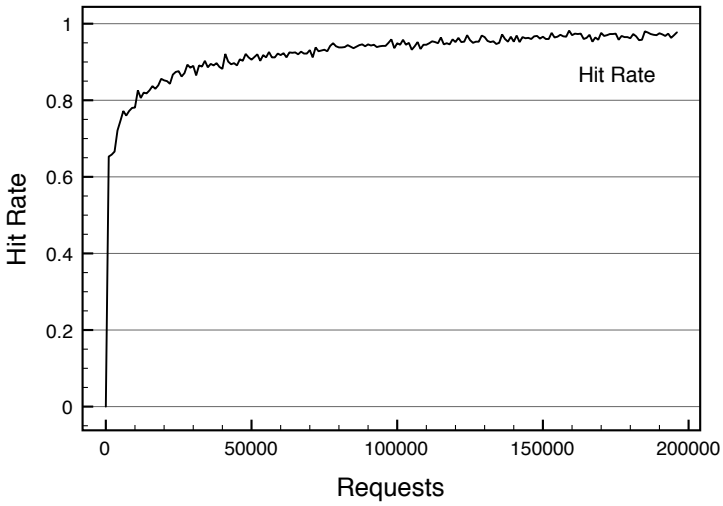
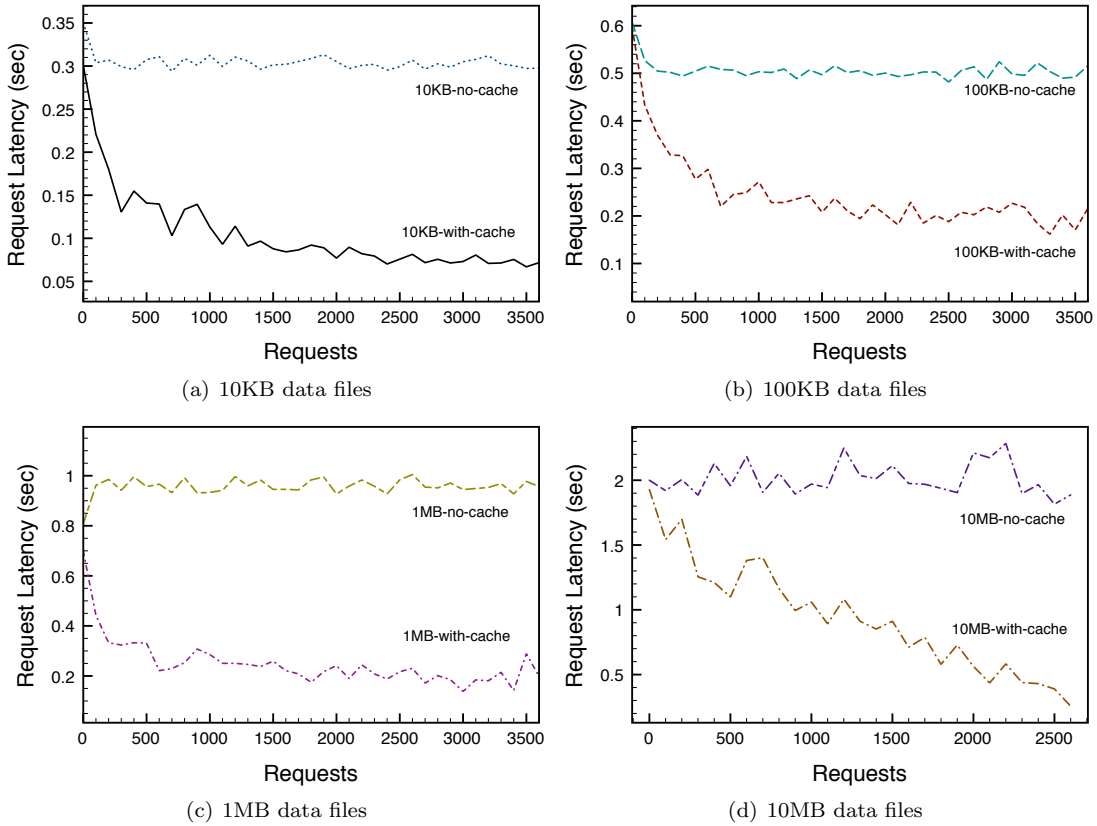


Figure 10. Hit Rate

We further analyze the average latency trajectory displayed by our system in Figure 11. 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 within specific file size ranges As seen previously in Figure 8(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 $\geqslant$ 100KB and $<$ 1MB. Figure 11 shows the query latencies observed per disaggregated data set, both with and without employing the cache. As expected, the latency drop is highest (close to 10× speedup) for the files ranging in magnitude from 1MB to 10MB. The charts in Figure 11 confirm that our cloud-based web cache can accelerate applications requiring large data movement for



(a) 10KB data files

(b) 100KB data files

(c) 1MB data files

(d) 10MB data files

Figure 11. Average Request Latency

data-intensive applications (note the scale on the y-axis has been selected to improve legibility). Next, we evaluate the cost and performance models.

4.3 Cost-Performance Model Evaluation

The mathematical model presented in Section 3 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 12 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).

Figure 13 highlights the results of a set of experiment with a constant query rate of 100 requests

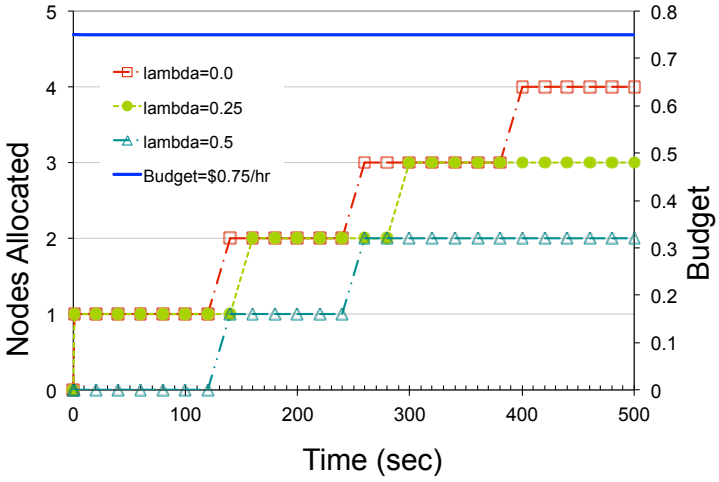


Figure 12. Optimal Node Allocation (Request Rate: 25 requests/sec)

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.

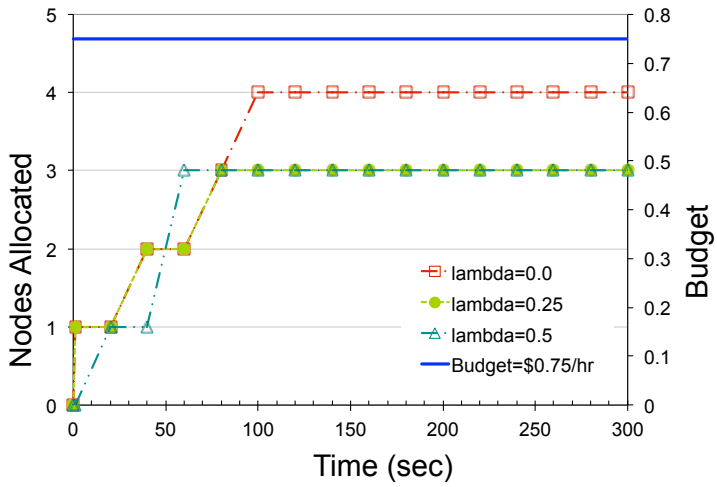


Figure 13. Optimal Node Allocation (Request Rate: 100 requests/sec)

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 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.

5. RELATED WORKS

This section summarizes the related works in the area of cloud resource allocation in order to achieve automated scaling.

Cloud computing has become a hot topic in both research and industry. According to Armbrust, *et al.*, developers who have innovative new ideas are no longer bounded by the capital required for large hardware infrastructures [Armbrust et al. 2010]. Additionally, organizations can retrieve results just as fast as their programs scale, as costs are levied on a computational basis. The sheer amount of research into the Cloud has been tremendous over the past several years. Study includes such topics as: creating frameworks for elastic high performance applications [Pham et al. 2011], infrastructure engines such as *OpenNEbula* [Wang et al. 2008], job scheduling systems for cloud service providers[Dutta et al. 2012], and distributing indices for multidimensional data [Papadopoulos and Katsaros 2011] — each looking to leverage or support the power that elasticity grants.

Vaquero *et al.* investigate the current state of cloud scaling and theorize three levels of automatic scaling that they consider imperative in an *ideal* elastic cloud [Vaquero et al. 2011]. In the context of server level scaling, they argue that mere server scalability is inadequate if scaling logic is not extended to the load balancing tier. The authors also point out the lack of network level auto scaling in current cloud implementations. They suggest exploiting available technologies, such as *Network Description Language* (NDL)-based ontologies for expressing the required network characteristics, statistical multiplexing to optimize bandwidth usage *etc.*, to dynamically allocate bandwidth on-demand. Gong, *et al.* describe PRESS, which can be used by cloud service providers to predict future consumer demand to avoid significant SLA violations and the associated penalties [Gong et al. 2011]. Their application-agnostic load prediction model employs mathematical algorithms and signal processing techniques to discover a signature for workloads with repeating patterns. For applications without repeating patterns, PRESS uses a discrete-time Markov chain to build a short-term prediction. In contrast, our prediction model uses an Artificial Neural Network to capture potentially complex request trends.

Ramaswamy, *et al.* outline the architecture of a cooperative cache cloud situated in edge networks [Ramaswamy et al. 2005]. They focus on the mechanisms for efficient cooperation among caches and introduce dynamic hashing-based protocols for document lookup and a utility-based mechanism for placing data objects on different nodes within the cloud-based cache. Chiu, *et al.* leverage elastic cloud caches for accelerating web service computations [Chiu and Agrawal 2010; Chiu et al. 2010; 2011]. In this system (which provides the foundation for our own system), the cache operates in a cooperative manner and utilizes consistent hashing to assist in document lookup. They also introduce the notion of cache eviction and contraction, merging nodes as query traffic rate subsides. This system achieved 8× speedup by utilizing a cloud cache in support of web service-based scientific workflow applications.

Memcached [Fitzpatrick 2004] is a widely adopted, open-source, *key-value* caching system. Memcached is commonly used to cache small pieces of data (up to 1MB in size) in memory. The system is intended for speeding up web applications by caching arbitrary data resulting from database calls, API calls, or page rendering. In this way, *Memcached* reduces the back-end database system load. Also related, Amazon *ElastiCache* [Amazon Web Services Inc. 2013] is a pay-as-you-go service that leverages *Memcached* protocols. While users are able to manually scale their cache up and down (or impose certain rules to do so) to fit their requirements, it does not yet provide mechanisms for automating this process.

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 [Amazon EC2 2013]. Based on user defined policies on infrastructure-level performance metrics, AutoScaling essentially allows users to specify threshold values for certain metrics. Whenever the observed performance metric violates the given threshold, a predefined number of compute nodes are either added or removed from the appli-

cation's resource pool. However, in cases where much more distributed coordination is required, these mechanisms render themselves inadequate and elasticity does not directly translate to scalability. In contrast, our system implements a specific scaling logic by automatically migrating data objects to and from nodes as they scale. 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 [Mao et al. 2010; Mao and Humphrey 2011]. 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 [Shen et al. 2011]. 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 [Zhu et al. 2012]. 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.

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.

Another related area is the problem of web cache replacement policies. Podlipning and Böszörmenyi performed a survey and classification of a number of web cache replacement strategies [Podlipnig and Böszörmenyi 2003]. There, they classify replacement strategies on a number of factors: *recency*, *frequency*, *size*, *cost of fetching the object*, *modification time*, and a heuristic *expiration time*. They then perform a survey of different proposals for each of the different eviction strategies. Wong performs a survey over web cache replacement policies and categorize the available replacement policies [Wong 2006]. They also analyze the design considerations behind each of the categories, noting such things as *hit ratio*, and *complexity*. Additionally, they take a more pragmatic approach to their analysis and provide recommended replacement policies based on system characteristics and workload features. Reddy and Fletcher examine a more intelligent web caching algorithm using document life histories [Reddy and Fletcher 1998]. There they suggest that Least Recently Used (LRU) techniques are inadequate and they develop a mathematical model to predict the future value of cached files. The mathematical model they build makes use of damped exponential smoothing, utilizing file request frequency and the time since the last request. According to Psounis and Prabhakar, their randomized replacement scheme is used most efficiently whenever a large population of objects is accessed "most" frequently. However, as our cache is developed for service- and data-oriented web services, it is likely that data objects will follow predictable distributions [Psounis and Prabhakar 2001; Bhattacharjee and Debnath 2005].

6. CONCLUSION AND FUTURE WORK

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.

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 [Zhao and Schulzrinne 2003]. 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.

Our system currently only considered the FIFO and LRU replacement schemes. In the future, we would like to introduce a more cost-aware data placement and replacement scheme. In a budget-constrained system, we propose assigning *each data object* a score based on factors that are not limited to locality. For instance, some data objects may take a significant amount of time (translating to cost) to produce. Data objects can also vary wildly in size. In certain cases, we may want to demote a very large object down to S3 even if it is requested often. Prioritizing cache placement/replacement based on cost/performance metrics at the data object level holds significant potential for scalable cloud services.

REFERENCES

Amazon EC2. 2013. Amazon Auto Scaling. `http://aws.amazon.com/autoscaling/`.

Amazon Web Services Inc. 2013. Amazon ElastiCache. `http://aws.amazon.com/elasticache`.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. 2010. A view of cloud computing. *Commun. ACM 53,* 4 (Apr.), 50–58.

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

Barford, P. and Crovella, M. E. 1998. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of Performance '98/SIGMETRICS '98*. 151–160.

Bayer, R. and McCreight, E. 1970. 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*. ACM, New York, NY, USA, 107–141.

Bhattacharjee, A. and Debnath, B. 2005. A new web cache replacement algorithm. In *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on*. 420 – 423.

Bicer, T., Chiu, D., and Agrawal, G. 2012. 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)*.

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

Cardosa, *et al.*, M. 2011. Exploring mapreduce efficiency with highly-distributed data. In *MapReduce'11*. ACM, 27–34.

Chiu, D. and Agrawal, G. 2010. Evaluating caching and storage options on the amazon web services cloud. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing, Brussels, Belgium*.

Chiu, D., Shetty, A., and Agrawal, G. 2010. 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*. 1–11.

Chiu, D., Shetty, A., and Agrawal, G. 2011. Evaluating and optimizing indexing schemes for a cloud-based elastic key-value store. In *Proceedings of the 11th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE.

Das, S., Agrawal, D., and El Abbadi, A. 2009. Elastras: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*. HotCloud'09. USENIX Association, Berkeley, CA, USA.

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

Dutta, K., Guin, R., Banerjee, S., Chakrabarti, S., and Biswas, U. 2012. A smart job scheduling system for cloud computing service providers and users: Modeling and simulation. In *Recent Advances in Information Technology (RAIT), 2012 1st International Conference on*. 346 –351.

Eicken, T. V. 2008. The Rightscale Blog. `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/`.

Fitzpatrick, B. 2004. Distributed caching with memcached. *Linux J. 2004*, 5–.

Gong, Z., Gu, X., and Wilkes, J. 2011. Press: Predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th IEEE/IFIP International Conference on Network and Services Management, CNSM 2010, Niagara Falls, Canada*.

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

Karger, *et al.*, D. 1999. Web caching with consistent hashing. In *WWW'99: Proceedings of the 8th International Conference on the World Wide Web*. 1203–1213.

Lin, H., Ma, X., Archuleta, J., Feng, W.-c., Gardner, M., and Zhang, Z. 2010. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. ACM, New York, NY, USA, 95–106.

Mao, M. and Humphrey, M. 2011. 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*.

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

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

Nissen, S. 2003. Implementation of a fast artificial neural network library (fann). Tech. rep., Department of Computer Science University of Copenhagen (DIKU). http://fann.sf.net.

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

Papadopoulos, A. and Katsaros, D. 2011. A-tree: Distributed indexing of multidimensional data for cloud computing environments. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. 407 –414.

Pham, T. V., Truong, H.-L., and Dustdar, S. 2011. Elastic high performance applications – a composition framework. In *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*. 416 –423.

Podlipnig, S. and Böszörmenyi, L. 2003. A survey of web cache replacement strategies. *ACM Comput. Surv. 35,* 4 (Dec.), 374–398.

Psounis, K. and Prabhakar, B. 2001. A randomized web-cache replacement scheme. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. 1407 –1415 vol.3.

Ramaswamy, L., Liu, L., and Iyengar, A. 2005. Cache clouds: Cooperative caching of dynamic documents in edge networks. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*. 229 –238.

Reddy, M. and Fletcher, G. P. 1998. Intelligent web caching using document life histories: A comparison with existing cache management techniques. In *In 3rd International WWW Caching Workshop*. 35–50.

Ruby Documents. Module:kernel (ruby 1.9.2). In *http://www.ruby-doc.org/core-1.9.2/Kernel.html#method-i-rand*.

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

Vaquero, L. M., Rodero-Merino, L., and Buyya, R. 2011. Dynamically scaling applications in the cloud. *Computer Communication Review 41,* 1, 45–52.

Wang, L., Tao, J., Kunze, M., Castellanos, A., Kramer, D., and Karl, W. 2008. Scientific cloud computing: Early definition and experience. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*. 825 –830.

Wong, K.-Y. 2006. Web cache replacement policies: a pragmatic approach. *Network, IEEE 20,* 1 (jan.-feb.), 28 –34.

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

Zhu, T., Gandhi, A., Harchol-Balter, M., and Kozuch, M. 2012. Saving Cash by Using Less Cache. In *HotCloud '12*. Boston, MA.

Farhana Kabir is a Software Engineer at Hewlett-Packard's Inkjet Print Solutions (IPS) Group working on Instant Ink, a worldwide ink subscription service. She received an M.S. in Computer Science from Washington State University Vancouver, where her research area was cloud computing, under the supervision of Dr. David Chiu. She also holds a B.S. in Computer Science from Purdue University. Her prior industry experience is with Oracle and Intel Corporations. Currently, she is serving as a Program Committee member for Data Analytics 2014.

Travis Hall is a Software Engineer working in identity management, web technology, and cloud computing for VMware. He graduated with a M.S. in Computer Science from Washington State University, Vancouver where he researched cloud computing under the supervision of Dr. David Chiu. In the past he has also worked for ForgeRock, an open source identity management stack and under Google's Summer of Code program on a web service designed to provide students with course- and assignment-based code respositories. Web technology, cloud computing, and Computer Science education are among some of his primary interests.

Scott Wallace received his Ph.D. in Computer Science from the University of Michigan in 2003. He is currently an Associate Professor of Computer Science at Washington State University Vancouver. His research interests include Artificial Intelligence, Machine Learning and Computer Science Education. He has recently received grants from Federal agencies (NSF) and State level organizations (Washington Technology Center, Oregon Best) for pedagogical and technical research.

David Chiu is an Assistant Professor and the CS Graduate Studies Chair in the School of Engineering and Computer Science at Washington State University, Vancouver. He received a Ph.D. in Computer Science and Engineering from The Ohio State University in 2010, where he worked with Prof. Gagan Agrawal in the Data Intensive and High-Performance Computing Group. His current research interests span data-intensive computing, cloud resource allocation, and data management. He is a member of the ACM, IEEE, and Upsilon Pi Epsilon.