2013 IEEE 33rd International Conference on Distributed Computing Systems

Proteus: Power Proportional Memory Cache Cluster in Data Centers

Shen Li, Shiguang Wang, Fan Yang, Shaohan Hu, Fatemeh Saremi, Tarek Abdelzaher
University of Illinois at Urbana-Champaign, USA
{shenli3, swang83, fanyang5, shu17, saremi1, zaher}@illinois.edu

***Abstract*—In this paper, we describe the design, implementation and evaluation of Proteus, a power-proportional cache cluster which eliminates the delay penalty during server provisioning dynamics. To speed up data center services, a cache cluster is used in front of the database tier, providing fast in-cache data access. Since the number of cache servers is large, building power-proportional cache clusters can lead to considerable monetary savings. Dynamic server provisioning, one common methodology for realizing power proportionality in data centers, calls for agile load balancing schemes and smart in-cache data migration algorithms when applied to cache clusters. Otherwise, it induces unacceptable delay spikes due to data re-allocation among cache servers. Proteus addresses both challenges by using a specifically designed virtual nodes placement algorithm and an amortized data migration policy. We implement Proteus, and evaluate it on a 40-server cluster using real Wikipedia data and workload traces. The results show that, with Proteus, the load distribution is much more evenly balanced compared to the case of applying unmodified consistent hashing. At the same time, Proteus induces almost no extra delay during provisioning transitions, which is a significant advantage over other state-of-the-art solutions.**

Index Terms: Energy proportionality, Load balancing, Memcached, Bloom filter, Data center.

I. Introduction

In this paper, we present Proteus, which aims at reducing performance penalties caused by applying energy management policies to memory cache clusters where cache servers may be turned on and off. Energy consumption in cloud computing attracts attention due to the increasing energy cost in large data centers. Most of the previous work [1–8] focuses on stateless or computation-intensive workloads. Some systems [9–11] manipulate the data layout in distributed file systems (DFS) to create opportunities for turning off a subset of data storage servers. However, the energy issue in cache clusters is surprisingly ignored. To the best of our knowledge, this is the first paper that achieves power proportionality in memory cache clusters without sacrificing performance.

The memory cache cluster usually sits in front of the database or distributed file system tier to offer fast in-memory data access by running Memcached instances. The cache layer is playing a very important role. For example, Facebook reported that their cache hit ratio is higher than 95% [12], which significantly reduces the database workload. (Below, we use cache and Memcached interchangeably.) In industry data centers, workload seen by the cluster varies over time, and the peak workload can be as much as twice the valley workload [13]. This property offers us golden opportunities to apply dynamic server provisioning policies, such that when the workload is light, a subset of the servers is turned off to save energy. As the cache cluster may consist of hundreds or thousands of servers[14], the monetary benefits brought by energy management could be large.

Dynamic server provisioning, a common energy management methodology, is widely used for both stateless web servers and distributed replicated file systems. Unfortunately, due to distinct characteristics of cache clusters, directly applying dynamic server provisioning will suffer from severe performance degradation during provisioning dynamics (i.e., when servers are turned on or off). For stateless web servers, a reasonable assumption is that one request can be handled by any server without much performance penalty. Nevertheless, this assumption does not hold for the cache cluster. If one request is directed to some cache server that does not have the requested data, the request will have to reach the database tier (or DFS), which induces high response time. For distributed File systems, requests are usually directed by using deterministic load distribution policies. Many current designs employ master server [15], name node [16], or meta servers [17] to store the data chunk location information. One request will first reach the master server to look up the corresponding chunk server address, and then communicate with the chunk server to fetch data. However, given that the cache servers are designed for fast in-memory data access for a large number of data items, similar look-up operations, which involve one or more disk seeks, are too slow to serve the cache tier.

Under static scenarios, achieving load balancing is trivial: The web server can simply hash the requested data ID to a large integer range, and then apply the modulo operator to wrap it to a valid cache server ID. Reddit, a popular social news website, did use this scheme before. However, hey soon experienced the pain of expanding cache clusters for capacity upgrades [18]. This is because, in an n-server cache cluster, if one more server is added, this simple solution will remap $\frac{n}{n+1}$ data IDs to cache servers where the requested data is not in-memory. Therefore, in expectation, $\frac{n}{n+1}$ of the requests will reach the database tier simultaneously, and the databases are easily overloaded. The same problem happens when dynamic server provisioning is used to turn servers on or off. Consistent hashing alleviates the problem to some degree. Nevertheless, how to balance load under dynamics and how to make dynamic server provisioning smooth enough are still problems that are yet to be solved.

In this work, we focus on achieving three performance objectives in the cache cluster when using dynamic server provisioning policies. *First, the load distribution among all running cache servers should be balanced.* No matter how

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.50

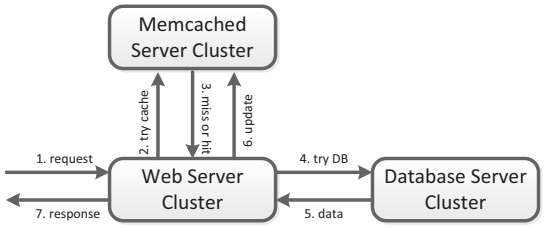

Fig. 1. Simplified Information Flow

many cache servers are running, each server should handle roughly the same amount of user workload. *Second, the provisioning transition should not induce much response time penalty.* Proteus needs to make this operation unnoticeable to users, although losing in-memory "hot" data and data migration for load re-balancing is inevitable when turning servers off. *Third, the load distribution algorithm should be distributed, consistent, and efficient.* As there are multiple web servers answering a large number of user requests, each web server should be able to make load distribution decisions locally. At the same time, the decisions must be consistent among different web servers, such that requests asking for the same piece of data are directed to the same cache server.

To achieve these objectives, Proteus presents algorithms for flexible load balancing and smooth provisioning transitions. We inherit the consistent hashing idea to achieve load balancing by providing a deterministic virtual node placement algorithm. All physical cache servers are guaranteed to handle the same amount of data IDs regardless of the size of the running cache cluster. Another merit of this algorithm is that the amount of data that needs to be migrated during each provisioning transition is at its lower bound. For smooth provisioning transitions, we modified Memcached source code [19] to insert a counting Bloom filter as the in-cache data digest. The digest is used by web servers to determine the current location of the requested data. Then the data migration cost is amortized over related requests. Our evaluation results show that Proteus balances the load distribution at the same level as the simple hash does in static cases, which is much more balanced than applying unmodified consistent hashing. Meanwhile, Proteus induces almost no delay penalty during dynamic server provisioning.

The remainder of this paper is arranged as follows. Section II briefly introduces the background and motivates this work. We present system design details in Section III, and Section IV. Implementation and evaluation are presented in Section V and Section VI, respectively. Related work is surveyed in Section VII, and finally, Section VIII concludes the paper.

II. System Model

In this paper, we aim at eliminating the performance penalties caused by dynamic cache cluster provisioning in a 3-tier server cluster as shown in Fig. 1. Time is divided into slots. Let N denote the total number of cache servers, and $n(t)$ denote the number of running cache servers in the t^{th} time slot. The data key key^d can be a page title in Wikipedia, a user ID in Facebook, etc. With key^d, the web server cluster fetches corresponding data d from either the cache server m^d or the database tier, and then presents the response to users. Cache servers store data in the form of $(key, data)$ pairs. We define a piece of data as "hot" if it is touched at least once during the past TTL seconds. Please note that, we do not make any assumptions on the cache eviction strategy (LRU, fixed expiration duration, etc.). At one time instance, a cache server is called *active* if it hosts "hot" data and serves requests. Otherwise, it is called *inactive*, and operates in low power status.

Assumptions

- Each object in cache is of the same size. Even though the size of pages or user accounts would vary considerably, they can be divided into fixed-size pieces. One piece is considered as the basic unit of objects in cache. Actually, modern storage clusters already employ such idea [15–17].
- The load of requests have temporal behavior, and the gap between the peak and the nadir load is huge. Our study of Wikipedia's traces supports this assumption (as shown in Fig. 4).

Objectives

- *Balance load distribution in cache tier under dynamics*: We define the load as the amount of data objects served by the cache cluster. In static environments, the load can be easily balanced by using hash and modular operations. We pursue the same level of load balance in face of dynamic server provisioning.
- *Minimize data movements during re-balancing transitions*: Proteus should achieve minimum data migration, such that only at most $\frac{|n(t+1)-n(t)|}{\max\{n(t+1),n(t)\}}$ of the in-cache data is remapped when the number of active servers changes from $n(t)$ to $n(t+1)$.
- *Eliminate re-balancing transition delay spikes*: Dynamic server provisioning should never hurt the performance too much. We aim at managing energy with no delay penalty.

Generally, our goal is to design a provisioning actuator that executes decisions according to server provisioning policy without degrading the system performance in terms of response time. Please note that designing the best provisioning policy is *not* our focus. Different policies [1, 4–8] can cooperate with Proteus with no confliction.

III. Load Balance under Dynamics

In this section, we describe an algorithm that not only deterministically balances load distribution under provisioning dynamics but also guarantees migrating minimum amount of data during each re-balancing transition. Our algorithm inherits ideas of consistent hashing and virtual nodes [20, 21], and focuses on generating a virtual node placement strategy.

Load balance under dynamics requires each server handles equal amount of objects even if the number of active servers dynamically changes. As already used in many Memcached clusters [22], consistent hashing and virtual nodes help to balance load among multiple cache servers to some degree [20, 21]. In consistent hashing, data keys and servers are hashed into the same key space, which forms a hashing ring. The ring acts as an indirect layer of index for the hash. Each obejct will be stored in the first server that succeeds the data key on the ring. When one server is turned off, its direct successor takes care of its workload. One physical server may have multiple direct successors by placing multiple virtual nodes on the hashing ring. Therefore, the load of one server can be spread out to more physical servers when necessary. However, without careful design, consistent hashing and virtual nodes alone do not guarantee load balance in face of provisioning dynamics. In this section, we borrow the idea of consistent hashing, and improve it by designing and analyzing a virtual node placement algorithm that deterministically balances workload among all active Memcached servers.

A. Fixed Provisioning Order

In data centers, every individual server is under control. Hence, it is possible to turn on/off servers according to any fixed order. We argue that, a provisioning scheme with a fixed order is not any weaker than the one that adapts to arbitrary orders. The reason is that, when no failure presents, the fixed order can be maintained easily. If some server crashes, we have already lost the data in cache, and both schemes need some fault tolerant solutions for reconstructing the cache or directing the requests to some redundant caches.

Fixing provisioning order eliminates one dimension of freedom of the load balance problem, thus simplifies the algorithm design. Well designed order further improves power savings. For example, the decreasing order of server efficiency should be better than a random order, where server efficiency is defined as the amount of workload served per unit of energy. The system administrators should be responsible for choosing a reasonable order, which is not the focus of this paper. Our solutions are able to cooperate with any fixed order.

Define the list $(s_1, s_2, ..., s_N)$ as the fixed order for dynamic provisioning, where N is the number of servers. Let $n(t)$ denote the number of active servers in the t^{th} time slot. In other words, servers in set $\{s_i \mid 1 \leq i \leq n(t)\}$ are active in the t^{th} time slot.

B. Optimal Number of Virtual Nodes

Let $\mathcal{V}_i = \{v_{i1}, v_{i2}, v_{i3}, ...\}$ denote the set of virtual nodes of the server s_i. Any request within the key range between the virtual node v_{ij} and its direct predecessor on the hashing ring will be served by virtual node v_{ij}, and thus, physical node s_i. We call this key range the *host range* of the virtual node v_{ij}. As the provisioning algorithm turns on and off physical nodes dynamically, the direct successor of v_{ij} varies as well. However, recall that the order for provisioning is static, v_{ij} always precedes the same direct successor if the number of active servers is the same. We define v_{ij}'s *final successor* as the direct successor when $i-1$ servers are on. Denote $v_{ik_i} \rightarrow v_{jk_j}$ as the relation between two virtual nodes v_{ik_i} and v_{jk_j}, such that v_{jk_j} is the final successor of v_{ik_i} on the consistent hashing ring. Let $\mathcal{P}_i^s$ denote the set of final successor servers of server s_i (as illustrated in Fig. 2), *i.e.*,

$$\mathcal{P}_i^s = \{s_j \mid \exists v_{jk_j} \in \mathcal{V}_j, \exists v_{ik_i} \in \mathcal{V}_i, \textit{ s.t. } v_{ik_i} \rightarrow v_{jk_j}\}.$$

In order to achieve load balance under provisioning dynamics, the virtual node placement policy should satisfy the following conditions: (1) when one physical node is turned off, objects served by this node should be evenly migrated to all other running physical nodes, and (2) when one physical node is turned on, it should take an identical amount of objects from all other active physical nodes. We call this condition the *Balance Condition (BC)*.

Obviously, among all solutions that satisfies BC, the one with minimum number of virtual nodes is preferred, since less virtual nodes introduce less overhead in terms of both space consumption and query complexity. We now prove that $\frac{N^2-N}{2}+1$ is the lower bound on the number of virtual nodes in order to meet BC.

Theorem 1: At least $\frac{N^2-N}{2}+1$ virtual nodes are required to satisfy BC.

Proof: We first introduce a necessary condition of BC, which we call the *pseudo BC*:

$$\mathcal{P}_i^s \supset \{s_j \mid 1 \leq j \leq i-1\}.$$

The pseudo BC states that to achieve load balance, the final successor set $\mathcal{P}_i^s$ should at least cover $s_j, \forall j < i$. Otherwise, if $\exists j < i, s_j \notin \mathcal{P}_i^s$, s_i's workload will not be directed to s_j when s_i is turned off. Hence the load is not balanced. Please note that the pseudo BC does not guarantee the feasibility of host ranges. [1].

One virtual node can only have one final successor on the hashing ring. Since $|\mathcal{P}_i^s| \geq i-1$, s_i needs at least $i-1$ virtual nodes to precede $i-1$ final successors. The corner case is $i=1$, where s_1 needs to have at least 1 virtual node. Therefore, altogether, at least $1+\sum_{i=2}^{N}(i-1) = \frac{N^2-N}{2}+1$ virtual nodes are required. ■

C. Virtual Node Placement

In this section, we elaborate the virtual node placement algorithm. Assume the key space size is $\mathcal{K}$. When generating the placement solution, physical nodes are served one by one according to the fixed provisioning order. For s_i $(i > 1)$, the algorithm places $i-1$ virtual nodes on the consistent hashing ring, denoted by $\{v_{ij} | 1 \leq j \leq i-1\}$. When placing v_{ij}, the algorithm borrows $\frac{\mathcal{K}}{i(i-1)}$ continues host range from

[1]Counter Example: Place the virtual nodes clockwise on the consistent hashing ring with the following order: $1, 2, 3, ..., n, 2, 3, 4, ..., n, ..., (n-3), (n-2), (n-1), n, (n-2), (n-1), n, (n-1), n$. The numbers are corresponding to physical server ID. When $n > 6$, it is not possible to have all responsible ranges positive.

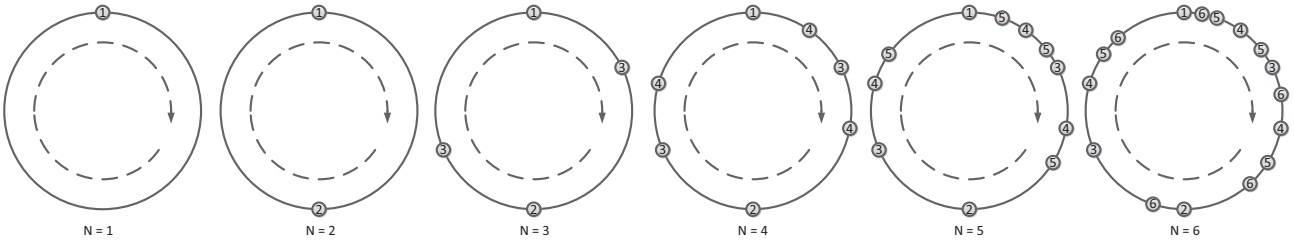

Fig. 2. An example of virtual node placement: Each small circle represents a virtual node. The number in the circle indicates the ID of the physical server that hosts the virtual node. The turned-on/off order is fixed as $(1, 2, 3, 4, 5, 6)$. Obviously, $\mathcal{P}_6^s = \{1, 2, 3, 4, 5\}$, $\mathcal{P}_5^s = \{1, 2, 3, 4\}$, $\mathcal{P}_4^s = \{1, 2, 3\}$,$\mathcal{P}_3^s = \{1, 2\}$, and $\mathcal{P}_2^s = \{1\}$. Since we are turning off servers based on the above order, server 6 is already powered-off when server 5 is being turned off. Hence, when calculating $\mathcal{P}_5^s$, server 6 no longer exists in the hashing ring.

one feasible virtual node of s_j, and assigns it to v_{ij}. Fig. 2 shows an illustrative example with 6 physical nodes. In Section III-D, we prove that the algorithm is correct and the host ranges are balanced among all physical nodes.

As each virtual node can be identified by its host range, placing virtual node is equivalent to assigning an unique host range to each virtue node. Below, we use host range and virtual node interchangeably. The pseudo code is shown in Algorithm 1. The algorithm takes 2 inputs: the number of physical nodes N, and the consistent hashing ring range $\mathcal{K}$. Line $2-3$ adds s_1's only virtual node $v_{1,0}$ into s_1's host range set $R[1]$. Initially, $v_{1,0}$ covers the entire consistent hashing ring. The loop starting on line 4 enumerates all $s_i, i > 1$ to add their virtual nodes. Line $5-15$ iterates to place each of s_i's virtual node v_{ij}. For v_{ij}, the inner loop on line 6 searches for one feasible virtual node r of s_j whose host range is larger than $\frac{\mathcal{K}}{i(i-1)}$. When found, v_{ij} borrows $\frac{\mathcal{K}}{i(i-1)}$ from r, and the algorithm inserts v_{ij} into s_i's virtual node set $R[i]$.

D. Algorithm Analysis

We prove the virtual node placement solution generated by Algorithm 1 balances the sum of host ranges assigned to each physical server.

Proof: **basis:** If $N = 1$, there is only one node. Hence the host range is trivially balanced.

Inductive Step: Assume the algorithm is correct for $N = k$ (*i.e.*, the host range of each physical node is $\frac{\mathcal{K}}{k}$). For $N = k + 1$, the host range of each physical node should be $\frac{\mathcal{K}}{k+1}$. Hence, to achieve load balance, the $(k+1)^{th}$ node needs to deduct $\frac{\mathcal{K}}{k(k+1)}$ from at least one virtual node of each physical node with smaller ID, which is done in line $6 \sim 14$ in Algorithm 1. We now show that it is always possible to find one feasible virtual node from $R[j]$ whose host range length is at least $\frac{\mathcal{K}}{k(k+1)}$. By contradiction, we assume that for some s_i, $(i < k + 1)$, the host range length of all $v_{ij}, (j < i)$ is smaller than $\frac{\mathcal{K}}{k(k+1)}$. Therefore, the sum of host range length deducted from s_i by physical nodes set $\{s_l | i < l < k + 1\}$ is at least at least

$$(i-1)\left(\frac{\mathcal{K}}{i(i-1)} - \frac{\mathcal{K}}{k(k+1)}\right). \quad (1)$$

According to the algorithm, each $s_l, l > i$ borrows $\frac{\mathcal{K}}{l(l-1)}$ from s_i. So, we also have, $\sum_{l=i+1}^{k} \frac{\mathcal{K}}{l(l-1)} = \frac{\mathcal{K}}{i} - \frac{\mathcal{K}}{k}$, which is smaller than formula (1),

$$(i-1)\left(\frac{\mathcal{K}}{i(i-1)} - \frac{\mathcal{K}}{k(k+1)}\right) - \left(\frac{\mathcal{K}}{i} - \frac{\mathcal{K}}{k}\right) = \mathcal{K}\frac{ik - i^2}{ik(k-1)} > 0. \quad (2)$$

Hence, the assumption does not hold. It is always possible to borrow $\frac{\mathcal{K}}{k(k+1)}$ host range length from at least one virtual node of each physical node s_i, $(i < k)$. Therefore, the host ranges are balanced in $N = k + 1$ case. ■

Algorithm 1 Virtual node placement

Input: Number of physical nodes N, consistent hashing ring range $\mathcal{K}$.
Output: Virtual node placement strategy vps.

1: $R \leftarrow$ an array of N empty set
   /* Host range set array for all physical vertices. */
2: $v_{1,0}.start \leftarrow 0$, $v_{1,0}.len \leftarrow \mathcal{K}$
3: $R[1] \leftarrow \{v_{1,0}\} \bigcup R[1]$
   /* Initially, the host range of s_1's only virtual node starts at 0 with length $\mathcal{K}$*/
4: **for** $i \leftarrow 2$ **to** n **do** /* enumerate all s_i */
5: **for** $j \leftarrow 1$ **to** i-1 **do** /* compute host range for v_{ij} */
6: **for** r **in** $R[j]$ **do**
   /* borrow host range from one feasible virtual node of s_j*/
7: **if** $r.len > \frac{\mathcal{K}}{i(i-1)}$ **then**
8: $v_{ij}.start \leftarrow r.start$, $v_{ij}.len \leftarrow \frac{\mathcal{K}}{i(i-1)}$
9: $r.len \leftarrow r.len - \frac{\mathcal{K}}{i(i-1)}$
10: $r.start \leftarrow r.start + \frac{\mathcal{K}}{i(i-1)}$
11: $R[i] \leftarrow \{v_{ij}\} \bigcup R[i]$
12: **Break**
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: $vps \leftarrow \emptyset$ /*serialize host ranges in a sorted array for fast access*/
18: **for** i $\leftarrow 1$ **to** N **do**
19: **for** r **in** $R[i]$ **do**
20: $vps \leftarrow vps \bigcup \{r\}$
21: **end for**
22: **end for**
23: sort vps based on $vps[\cdot].start$
24: **Return** vps

E. Fault Tolerance

Given the scale of data centers, server failure is not an exception, instead, it occurs frequently. For the sake of fault tolerance, Proteus can easily extend to embrace redundancies. For example, if Proteus is set to keep r replications for each piece of $(key, data)$ pair, it just needs to construct r consistent hashing rings with r different hash functions. Different hashing rings share the same virtual nodes placement policy. If a key falls in the host range of any virtual node of s_i on any hashing ring, s_i will store one copy of the $(key, data)$ pair. This strategy does not

guarantee that the r replications will be stored in r different servers. However, the confliction probability will be low. If the hash function distributes the keys evenly and randomly. The probability that no confliction occurs is:

$$P_{nc} = \prod_{i=0}^{r-1} \frac{n(t) - i}{n(t)}. \quad (3)$$

As r is usually a small number (*e.g.*, 2 or 3), and $n(t)$ is much larger (*e.g.*, a few thousand), P_{nc} for each data piece should be close to 1.

IV. Smooth Provisioning Transition

In this section, we elaborate an efficient solution for smooth provisioning transition, which is executed before turning on/off any cache server.

Each Memcached server may host tens or hundreds of gigabyte "hot" data [23]. If we turn off the Memcached servers brutally, we will lose a considerable amount of in-cache data. As consequences, some users might see delay spikes. We propose *Smooth Provisioning Transition* to avoid such spikes. Our high level idea is that when turning off s_i, s_i postpones this operation by TTL seconds to migrate "hot" data to its final successor on demand. Define $\mathcal{H}_t$ as the consistent hash in t^{th} time slot. Let key^d and m_t^d denote the data key and the corresponding Memcached server ID for data d in t^{th} time slot. Since (key^d, d) may reside in both m_t^d and m_{t+1}^d during the transition stage, web servers need to know which cache server they should query.

Three objectives must be satisfied. First, the transition must be unblocking. Memcached is designed as a fast key value store. The performance of the Memcached tier should not be interfered too much, otherwise the solution is useless. Second, only the "hot" data should be transferred, otherwise bandwidth and computational resources are wasted. Third, the transition delay should be small and bounded. The number of Memcached servers is tuned to catch up with load dynamics. Thus, long transition delay harms the system agility.

To achieve these objectives, we propose to use one Counting Bloom Filter [24, 25] as the content digest for each cache server. Below, we elaborate the details of maintaining and configuring the counting Bloom filter.

A. MemCached Digest

Bloom filter is a data structure that stores a set of elements and supports membership queries. Counting Bloom filter is a variant of Bloom filter that supports both element insertion and deletion. In our solution, each Memcached server maintains one counting Bloom filter for in-cache keys. We call the counting Bloom filter the $digest$. Let $\mathcal{F}_i$ denote the $digest$ of s_i. When key^d and its data are inserted into (or deleted from *resp.*) s_i, $\mathcal{F}_i$ will also be updated accordingly. We do not assume any cache replacement policy, as long as $\mathcal{F}_i$ is consistent with s_i's content, our solution works.

At the beginning of the transition stage, digests (a few KB each) will be broadcasted to all web servers. Then, the web server knows what is in-cache and what is not. The algorithm for data retrieval is described in Algorithm 2. Line 2 $\sim$ 4 checks whether the data is in $s_{m_{t+1}^d}$. If hit, the algorithm returns the data and does not go any deeper. If miss, line 6 checks whether the data resides in $s_{m_t^d}$. If yes, it retrieves the data. Since Bloom filter may have false positives, line 8 further checks if the data is indeed retrieved. If and only if both attempts miss, will the request reach the database tier. Therefore, the database tier will not realize transition dynamics is taking place. If the data is retrieved from either $s_{m_t^d}$ or the database tier, the algorithm also updates $s_{m_{t+1}^d}$.

Algorithm 2 Date Retrieval

```
1: procedure FETCH_DATA(key^d)
2:     data ← s_{m^d_{t+1}}.get(key^d)
3:     if NULL ≠ data then
4:         return data                                  /* found in new server. */
5:     else
6:         if F_{m^d_t}.check(key^d) then               /* data is "hot". */
7:             data ← s_{m^d_t}.get(key^d)
8:         end if
9:         if NULL = data then      /* false positive or "cold" data. */
10:            data ← database.get(key^d)
11:        end if
12:        s_{m^d_{t+1}}.put(key^d, data)
13:        return data
14:    end if
15: end procedure
```

The algorithm has two important properties. First, for "hot" data d, only the first request will reach $s_{m_t^d}$, all subsequent requests will find the data in $s_{m_{t+1}^d}$. Therefore, no bandwidth and computational resources are wasted. Second, Memcached servers can be safely turned off after TTL seconds. Because, if one piece of data d is touched in the past TTL second, it has already been transferred to $s_{m_{t+1}^d}$. If it is not touched in the past TTL second, it is no longer "hot", and can be safely discarded.

B. Bloom Filter Configuration

The Bloom filter employs a bit array with h hash functions. All bits in the array are initialized to 0. When an element e_i is inserted, bits at position $hash_1(e_i)$, $hash_2(e_i)$, ..., $hash_h(e_i)$ will be set to 1. When querying the membership of e_i, the Bloom filter checks the bits at position $hash_1(e_i)$, $hash_2(e_i)$, ..., $hash_h(e_i)$. If all of them are 1, the Bloom filter answers "yes". Otherwise, it answers "no". Bloom filter may have false positives.

Counting Bloom filter is a variant of Bloom filter, which uses counters rather than bits. When one element is inserted/deleted, the corresponding counters will increase/decrease by 1. The Counting Bloom filter suffers from both false positive and false negative issues. False negative is caused by either deleting an absent element (due to false positive) or counter overflow. In our scenario, the first case will never happen. The deletion from $digest$ is only triggered by the deletion from Memcached. The Memcached deterministically knows whether an element is in-cache or

Symbol	Description
h	Number of different hash functions
κ	Number of inserted keys
l	Number of counters in Bloom filter
b	Number of bits in each counter

TABLE I
BLOOM FILTER PARAMETERS AND DESCRIPTIONS

not. Therefore, deleting absent element from $digest$ will never happen. However, counter overflow may occur. In order to achieve low false negative rate, the Bloom filter should curb its counter overflow probability. Otherwise, if counter overflows, underflow will also be possible, which triggers false negatives. The counter overflow probability decreases with the increase of the counter size and the number of counters. However, more and larger counters lead to more memory consumption and higher overhead when broadcasting $digest$s. We now compute Bloom filter configurations to achieve minimum memory consumption subject to given false positive and false negative rate constraints (p_p, p_n). Table I shows the descriptions of symbols.

The probability that one counter remains 0 after inserting κ keys into l counters with h hash functions is $(1-1/l)^{\kappa h}$. Hence, the false positive rate is,

$$(1-(1-1/l)^{\kappa h})^{\kappa} \approx (1-e^{-\frac{\kappa h}{l}})^{h}. \tag{4}$$

As Memcached is designed as a high performance software, fewer hash functions are preferred. Therefore, we take h as a parameter.

As we have stated above, the counter overflow (and hence, underflow) is the only reason of false negatives in our system. The probability that any counter is greater than 2^b is [25, 26],

$$\mathbf{Pr}\left(\max(counter) \geq 2^b\right) \leq l\binom{\kappa h}{2^b}\frac{1}{l^{2^b}} \leq l\left(\frac{e\kappa h}{2^b l}\right)^{2^b} \tag{5}$$

Let $G_p(l) = (1-e^{-\frac{\kappa h}{l}})^h$ and $G_n(l,b) = l(\frac{e\kappa h}{2^b l})^{2^b}$ represent the false positive and false negative rate respectively. Then, our objective is:

$$\begin{aligned} &Minimize && lb \\ &s.t. && G_p(l) \leq p_p \quad G_n(l,b) \leq p_n \end{aligned} \tag{6}$$

where p_p and p_n are given false positive and false negative probability bounds.

Take partial derivatives of $G_n(l,b)$ with respect to l, and b respectively, we have

$$\begin{aligned} \frac{\partial G_n(l,b)}{\partial b} &= \mathcal{C}\left(\frac{b}{2}\ln\frac{e\kappa h}{2^b l} - \ln 2\right) < \mathcal{C}\left(\frac{b}{2}\ln\frac{e\kappa h}{2^b l}\right) \\ \frac{\partial G_n(l,b)}{\partial l} &= G_n(l,b)\frac{1-2^b}{l} > -\mathcal{C}\frac{1}{l} \end{aligned} \tag{7}$$

where $\mathcal{C} = G_n(l,b)2^b$. When

$$\frac{bl}{2} > \left(\ln\frac{2^b l}{e\kappa h}\right)^{-1} \tag{8}$$

we have

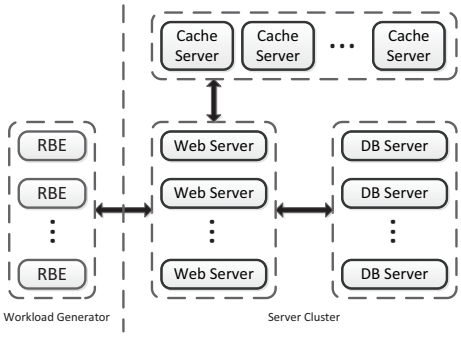

Fig. 3. System Setup

$$\frac{\partial G_n(l,b)}{\partial b} < \frac{\partial G_n(l,b)}{\partial l}. \tag{9}$$

Please note that 8 is almost always true, since $2^b l$ need to be much larger than $e\kappa h$ to achieve low false positive and false negative rate. Therefore, fixing lb, $G_n(l,b)$ decreases with the decrease of l. Hence, the optimal point is reached at the minimum possible l, which can be derived from the inequality $G_p(l)$. Hence, the optimal solution is

$$l = \frac{-\kappa h}{\ln\left(1-p_p^{\frac{1}{h}}\right)}, \quad b = \ln\left(\beta e^{\mathcal{W}\left(-\frac{\ln\gamma}{\beta}\right)}\right) \tag{10}$$

where $\beta = \frac{e\kappa h}{l}, \gamma = \frac{p_n}{l}$, and the function $\mathcal{W}(x)$ is the inverse of xe^x (Lambert function[27]).In practical, b is an integer with a very small range. Therefore, we can enumerate all possible values of b and pick the optimal one. For example, with ($\kappa = 10^4$, $h = 4$, $p_p = p_n = 10^{-4}$), ($l = 4 \times 10^5, b = 3$) is more than enough, which takes about $150KB$ memory per digest.

V. IMPLEMENTATION

In this section, we show the data and methodology details used in the evaluation.

A. System Setup

We use 40 Dell PowerEdge R210 servers with CentOS 5 (2.6.18 kernel) to build up the evaluating environment. RBE cluster consists of 10 servers who generate user workload. Another 10 Servers are used as web Servers by running our Java servlets on Tomcat 6.0.35. Memcached cluster has 10 servers running our modified Memcached release based on the source code pulled from github [19, 28]. The database cluster stores 2011-12-01 wikipedia-English dumps [29] in 7 non-overlapping shards on 7 different servers by running MySQL 5.5.19. Our cluster is equipped with Avocent PM3000 Power Distribution Unit which is able to measure the power reading of every single socket by SNMP. All 40 machines are connected by one NetGear Gigabit Smart Switch as a complete graph. The cluster is arranged as shown in Fig. 3.

1) Workload Generator: We evaluate Proteus with two different kinds of user workload: synthetic workload and real wikipedia trace [30]. For the synthetic workload, we simulate a large number of independent users. The user session duration follows exponential distribution. Each user has an independent, randomly selected wikipedia page set to access. The think time for each user is set to 0.5 second, so that we can simulate large number of requests with fewer number of threads. The total number of active users is dynamic and based on wikipedia trace. The wikipedia trace [30] logs the time and requested URL of every single access. However, the requested URL contains a lot of contents, such as images, which are not available to us. Hence, basing on the amount of request in Wikipedia trace, we use synthetic workload to evaluate cluster provisioning and smooth transition. As the data key is embedded inside the URL, we use the real Wikipedia trace [30] to evaluate load balancing performance and hit ratio of the counting Bloom filter.

2) Web Server: The servlets are running above Tomcat 6.0.35. Most logics are done in web servers, including hash data ID to Memcached server ID by using the consistent hashing as described in Section III, find the right mysql server if Memcached misses, check data availability by using counting Bloom filter as described in Section IV (counting Bloom filter is also implemented on Memcached servers), and so on. In order to reduce the overhead of creating new connections with Memcached server and database server, we use *Apache Commons Pool* to hold connection pools. Both connection pools are implemented as Singletons, such that all servlet threads will see the same connection pool.

3) Memcached with Build-in Bloom Filter: We modified Memcached [19] source code to enclose a built-in counting Bloom filter. The counting Bloom filter will insert key when $do\_item\_link(item *, const\ uint32\_t)$ function is called with key key, and will eject key when $do\_item\_unlink(item *, const\ unit32\_t)$ is called with key key. Memcached itself provides DTrace [31] probes which will notify user-defined programs when one specified function is called. This nice tool significantly simplifies the modification towards Memcached. Developers can define new logics without understand Memcached implementation details. However, we directly modify Memcached source code instead of using DTrace for two reasons. First, given that do_item_link or do_item_unlink can be touched hundreds of times per second, using DTrace will induce considerable overhead, since every time one Memcached DTrace probe fires, the context will switch between user mode and kernel mode twice. Second, as having one built-in Bloom filter in Memcached servers is meaningful, we hope our code is reusable to other researchers. Modifying the source code directly simplifies the usage.

Key "SET_BLOOM_FILTER" and "BLOOM_FILTER" are reserved for Bloom filter maintainable. When the client calls $get(\text{“}SET\_BLOOM\_FILTER\text{”})$, the Memcached server will take a snapshot of current Bloom filter bit array. Calling $get(\text{“}BLOOM\_FILTER\text{”})$ will retrieve the bit array as normal data. It exactly follows Memcached protocol, and should be compatible with all Memcached client package. We have conduct extensive tests with spymemcached [32] (implemented in Java), and Python-Memcached [33]. The results show that our Memcached release works perfectly with the above two clients.

Scenario	Server Provisioning	Workload Distribution
Static	All servers are on	simple hash with modular
Naive	Dynamically tuned	simple hash with modular
Consistent	Dynamically tuned	Consistent hashing
Proteus	Dynamically tuned	Proteus's algorithms

TABLE II
BLOOM FILTER PARAMETERS AND DESCRIPTIONS

4) Database: The Wikipedia English dump [29] is about 70GB in total. We divide the data horizontally into 7 pieces and have one MySQL server for each piece. The table engine is InnoDB and indexes are established for each key column. Each database request contains one single $page\_id$ parameter, which is used to look up $page\_latest$ from page table. Then, the MySQL will perform another $select$ in rev_text_id table by using the $page\_latest$ value, and finally, the rev_rext_id will be used to find the old_text value from $text$ table. The data retrieved from the corresponding old_text column will be sent back to web server, and the web server presents the data to RBE.

VI. EVALUATION

In the evaluation part, we compare four different scenarios: Static, Naive, Consistent, and Proteus. The detailed description is shown in Table II.

In this paper, we attack the performance penalty when applying dynamic cluster provisioning. For the sake of fairness, we use the same cluster provision feedback loop. Due the limit of space, the details of the feedback loop is omitted here. We run the feedback control algorithm along with Proteus with the delay bound set to 0.5 second. The feedback loop reference point is set to 0.4 second to tolerate overshot. The loop updates its status every 30 minutes. After this experiment, we know the number of running cache servers assigned to each 30-minute time slot, which is shown as the curve with small circles in Fig. 4. Here we use the number of requests as the workload based on which we do dynamic provisioning. It is true that the number of data being visited should be the real workload, because the bottleneck resource in Memcached cluster is the main memory. However, although the number of requests is not strictly linear proportional with the number of data being visited, it is a reasonable estimation and it is also easy to get. Please also note that our major focus is not designing optimal provisioning algorithm but the load balancing and smoothness of transition during provisioning dynamics. Then we apply the same cluster provisioning result, Wikipedia data and Wikipedia workload to all 4 different scenarios. In this way, the only differences of 4

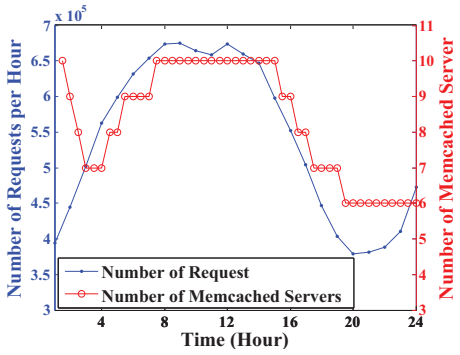

Fig. 4. WikiPedia Workload

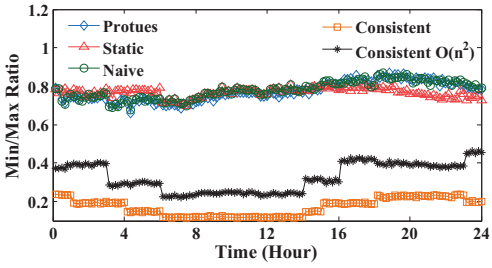

Fig. 5. Load Balancing

scenarios are the load balancing algorithm and their behavior in face of provisioning transition.

A. Load Balancing under Dynamics

We first evaluate how our load balancing algorithm works with real Wikipedia workload [30]. The trace contains timestamp and requested URL for every single user request. We first do some preliminaries to distill the requests that hit English Wikipedia. Then we sort all the request by timestamp, and count the number of requests inside every 1-hour time window. The result workload is depicted as the curve with dots in Fig. 4. Due to the lack of Wikipedia image data, our system cannot serve the real workload, and hence the response time for real workload is not measured. However, given that we are evaluating whether the load is balanced or not under dynamics in Memcached tier, we do not actually need the real image data. All we need is to measure whether the number of requests handled by each Memcached server is roughly the same. Again, as we stated before, the paper does not focus on how to build an efficient and accurate feedback loop for server provisioning. Here, we use the provisioning result as shown as the curve with circles in Fig. 4.

In Fig. 5, we study the performance of load balancing of the algorithms by depicting the ratio of $min\{L_i^t | i < n(t)\}$ over $\max\{L_i^t | i < n(t)\}$ for all t, where L_i^t is the workload on Memcached server i in t^{th} time slot. The curve with dimonds shows the load distribution by using Proteus, while the curves with small circles and triangles show the load distribution by using naive and static solutions (hash and modular) respectively. Clearly, Proteus achieves as good performance as the static and naive solutions. The curve

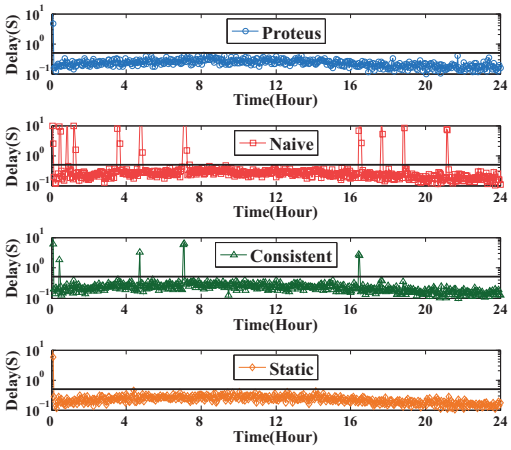

Fig. 9. Response Time

with squares depicts the load distribution using consistent hashing with $\mathcal{O}(\log n)$ randomly placed virtual nodes. The result is much worse than Proteus in face of real Wikipedia workload. Then we increase the number of virtual nodes for Consistent to $\frac{n^2}{2}$, and the result is shown by the curve with stars. It is better than the $\mathcal{O}(\log n)$ case but is still much worse than Proteus.

B. False Positive and False Negative

The counting Bloom filter is the key feature used as a digest for the Memcached data in the smooth transition phase. Both Memcached server and Bloom filter should be configured carefully, so that the cache achieves high hitting rate and the Bloom filter has low false positive and false negative rate. We apply the real Wikipedia trace [30] to evaluate Bloom filter settings. Fig. 6 shows how does the Memcached cache size affect hit ratio. When each Memcached server uses 1GB memory (with 4KB data per page), the hit ratio reaches above 80% (*i.e.*, roughly 2,560,000 pages in cache). With this setting, we proceed to tune the Bloom filter. As we stated before, using more hash functions in the Bloom filter induces higher overhead. Therefore, we choose to use only 4 non-encryption hash functions and tune the Bloom filter size instead. Fig. 7 and Fig. 8 present the relationship between the Bloom filter size and the false positive/negative ratio. As shown, with 512KB memory, the Bloom filter achieves negligible false positive and false negative rate. So, we set the Bloom filter to use 512KB memory when doing other evaluations.

C. Response Time

In this section, we will evaluate the service response time by applying synthetic workload to the server cluster. The response time is measured on the RBE side, which records the time duration from submitting one HTTP request to receiving the corresponding response. Each RBE server simulates hundreds of independent users with think time 0.5 second. As we have 10 RBE servers, altogether, there will be approximately a few thousand user requests per second. Each user has an independent page set of 50 pages. Every

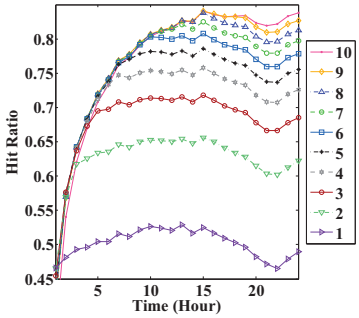

Fig. 6. Hit Ratio vs Cache Size

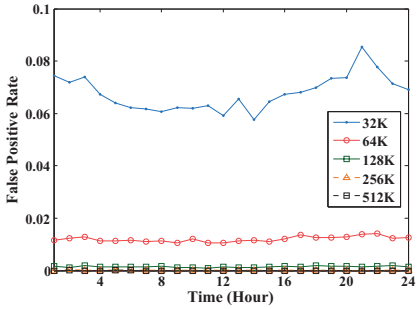

Fig. 7. False Positive vs Bloom Filter Size

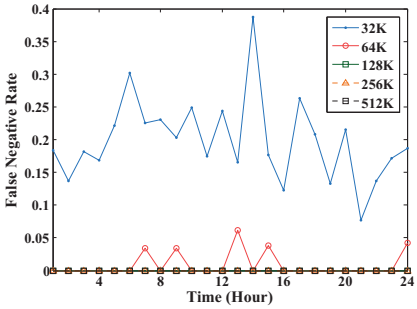

Fig. 8. False Negative vs Bloom Filter Size

time generating one request, the user thread will choose one page from her page set, and contact the web server to perform server side logic. The user requests will be uniformly randomly directed to all web servers.

Fig. 9 shows the experiment result. The recorded response time data are grouped into 480 slots according to physical time. We look at the response time locating at 99.9% percentile. The delay is plotted logarithmically. The curve with squares shows the response time for the Naive solution where the requests are directed by using simple hash and modular operation. Obviously, there is a huge response time spike when the number of running Memcached servers changes. As we explained before, it is because, when the number of running Memcached server changes, the mapping between data key and Memcached server ID changes significantly. A large number of requests will see misses in cache tier and will reach the database tier, hence induce a spike in response time. The curve with triangles depicts the response time when using consistent hashing with exactly $\frac{n^2}{2}$ virtual nodes. The virtual nodes are placed randomly using Java Random class. All web servers share the same random seed (0) to generate virtual node positions. Therefore, the view of all web servers are consistent. The consistent hashing solution shows much better performance during dynamics. But there are still considerable performance degradation during transition. The curve with circles demonstrates the response time when using Proteus. The delay spike is clearly removed, and users see almost no difference during the transition stages. Proteus's performance match what the static solution achieves as shown by the curve with diamonds.

D. Power Consumption

In previous sections, we have shown that, by using Proteus, the performance degradation induced by dynamic server provisioning is almost eliminated. Now, we compare the power consumption of the four different scenarios. We measure the real power reading by using the Avocent 3000 Power Distribution Unit. The data is sampled every 15 seconds. We take the entire cluster (web servers, cache server, and database servers) into consideration to see total power saving, rather than considering only the Memcached tier. Fig. 10 shows the power consumption over time. The Static scenario consumes roughly the same level of power

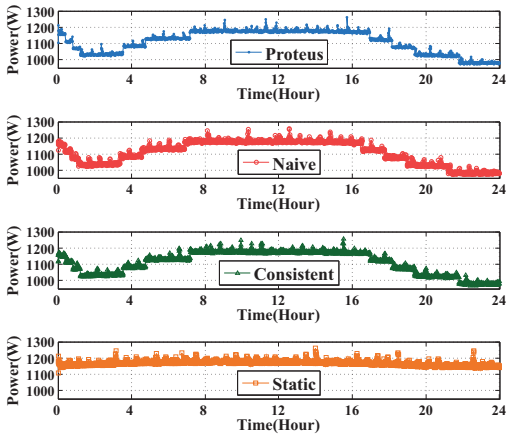

Fig. 10. Power Consumption

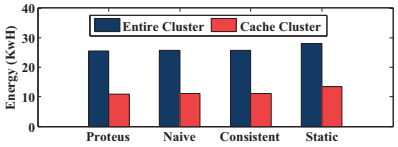

Fig. 11. Total Energy

during the whole experiment. The power consumption actually decreases slightly as the workload decreases. However, when compared with the power saving induced by server provisioning (as shown in the other three scenarios), the difference is almost unnoticeable. The curve with small dots shows the power draw of Proteus. It is clear that, Proteus not only eliminates performance degradations, but also saves the same amount of energy compared to Naive and Consistent cases. The total energy consumption during the experiments is shown in Fig. 11. The final result is that, with Proteus, we are able to save roughly 10% energy over the entire cluster, and 23% over the cache cluster without delay penalty.

VII. Related Work

Most previous data center energy management work fall into two scenarios: stateless clusters and stable storage systems. Plenty of prior work discussed [1–8] dynamic server provisioning under computational-intensive workload or stateless web servers. In these work, response time constraints or job deadlines are usually presented. The

OptiTuner [1] models 3-tier web server energy management problem as a optimization problem that aims at minimizing the energy consumption while follows the delay constraints. Decomposition technologies are used to achieve optimal iteratively. Yao *et al.* [2] and Liu *et al.* [3] try to minimize the cost among multiple geologically distributed data centers. Literature [4–7] takes the thermal effect into consideration to further improve the energy efficiency. However, all of these works focus on computation-intensive workload or assumes that the servers are stateless. As we have stated before, the assumption does not hold for many real systems (*e.g.*, web servers with session enabled, or cache clusters).

Another hot topic is energy management in distributed file system (DFS). Different from the stateless web servers, the most import property in DFS is data availability. Hence, related work [9–11] often focus on designing smart data placement policies, such that a portion of DFS servers can be powered off. In DFS, data are usually divided into small pieces and stored on different servers with multiple replications. In order to turn off DFS servers while guarantee data availability, Leverich *et al.* [10] proposed the covering set policy that one replication of each piece of data will be placed inside a small subset of DFS cluster. Hence, the remaining servers can be turned off safely without breaking the data availability guarantee. The GreenHDFS [11] further improve the idea by accounting data popularity. However, the above two papers do not consider load balancing the DFS system. Hrishikesh *et al.* [9] proposed the "equal work" policy, such that every running DFS server will serve the equal amount of data pieces regardless of the size of running DFS cluster. The above ideas are smart and reasonable. But they are not applicable to the cache cluster, as they rely on meta servers to store the location of every piece of data. The memory cache cluster are design for fast data access for huge amount of data pieces. Performing multiple operations on centralized meta servers for every single request will create a performance bottleneck, and hence, induce unacceptable huge delay.

VIII. Conclusion

In this paper, we attack the performance penalty caused by dynamic cache cluster provisioning in data centers. Tuning the the size of the cache cluster brutally leads to loss of in-memory "hot" data. Also, if load balancing is required, a significant amount of data will be re-mapped to different cache servers. Proteus proposes solutions for both problems. To minimize the amount of migrated data during load re-balancing, we design a virtual node placement algorithm that guarantees the amount of re-mapped data meets the lower bound. In order to make losing "hot" data less painful, we propose an on-demand data migration policy to amortize the cost to every single related request. By evaluating Proteus with Wikipedia data and workload trace on a 40-node cluster, we conclude that Proteus is able to reduce the cache cluster energy consumption without sacrificing response time.

References

[1] J. Heo, P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, and X. Liu, "Optituner: On performance composition and server farm energy minimization application," in *FeBID*, 2009.

[2] J. Yao, X. Liu, W. He, and A. Rahman, "Dynamic control of electricity cost with power demand smoothing and peak shaving for distributed internet data centers," in *ICDCS*, 2012, pp. 416–424.

[3] Z. Liu, M. Lin, A. Wierman, S. Low, and L. Andrew, "Greening geographical load balancing," in *ACM SIGMETRICS*, 2011, pp. 193–204.

[4] C. Bash and G. Forman, "Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *2007 USENIX Annual Technical Conference*, 2007, pp. 29:1–29:6.

[5] J. Moore, J. Chase, P. Ranganathan, and R. Sharma, "Making scheduling 'cool': Temperature-aware workload placement in data centers," in *USENIX Annual Technical Conference*, 2005, pp. 61–74.

[6] S. Li, T. Abdelzaher, and M. Yuan, "Tapa: Temperature aware power allocation in data center with map-reduce," in *IEEE International Green Computing Conference*, 2011, pp. 1 – 8.

[7] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher, "Joint optimization of computing and cooling energy: Analytic model and a machine room case study," in *IEEE ICDCS*, 2012.

[8] S. Li, T. Abdelzaher, S. Wang, M. Kihl, and A. Robertsson, "Temperature aware power allocation: An optimization framework and case studies," *Sustainable Computing: Informatics and Systems*, 2012.

[9] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," ser. SoCC, 2010, pp. 217–228.

[10] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 61–65, Mar. 2010.

[11] R. T. Kaushik and M. Bhandarkar, "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster," ser. HotPower, 2010, pp. 1–9.

[12] "Strategy: Break up the memcache dog pile," http://highscalability.com/strategy-break-memcache-dog-pile, August 2009.

[13] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: facebook's photo storage," ser. OSDI, Berkeley, CA, USA, 2010, pp. 1–8.

[14] L. Kai, A. Sridhar, S. Kannan, and S. Maguluri, "Indra: A data placement and replication system for online social networks," Technical Report, 2011.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," ser. SOSP, 2003, pp. 29–43.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10, 2010, pp. 1–10.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06, 2006, pp. 307–320.

[18] "And a fun weekend was had by all," http://blog.reddit.com/2010/03/and-fun-weekend-was-had-by-all.html, June 2012.

[19] "Memcached source code," git://github.com/memcached/memcached.git, April 2012.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," ser. SIGCOMM, 2001, pp. 149–160.

[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," ser. STOC, 1997, pp. 654–663.

[22] "Memcached internals," http://www.adayinthelifeof.nl/2011/02/06/memcache-internals/, November 2012.

[23] P. Saab, "Scaling memcached at facebook," http://www.facebook.com/note.php?note_id=39391378919, December 2008.

[24] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[25] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, pp. 281–293, Jun. 2000.

[26] G. H. Gonnet and R. Baeza-Yates, *Handbook of algorithms and data structures: in Pascal and C (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[27] J. H. Lambert, *Observationes variae in mathesin puram*. Acta Helvetica, physico-mathematico-anatomico-botanico-medica, 1758.

[28] "Memcached: A distributed memory cache system," http://memcached.org/, April 2012.

[29] "English wikipedia data dump," http://dumps.wikimedia.org/enwiki/, April 2012.

[30] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, pp. 1830–1845, July 2009.

[31] R. McDougall, J. Mauro, and B. Gregg, *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[32] "spymemcached," http://code.google.com/p/spymemcached/, April 2012.

[33] "Python-memcached," http://www.tummy.com/Community/software/python-memcached/, April 2012.