Design and implementation of caching services in the cloud

G. Chockler
G. Laden
Y. Vigfusson

Data caching is a key paradigm for improving the performance of web services in terms of both end-user latency and database load. Such caching is becoming an essential component of any application or service designed for the cloud platform. In order to allow hosted applications to benefit from caching capabilities while avoiding dependence on explicit implementations and idiosyncrasies of internal caches, the caching services should be offered by a cloud provider as an integral part of its platform-as-a-service portfolio. We highlight various challenges associated with supporting cloud-based caching services, such as identifying the appropriate metering and service models, performance management, and resource sharing across cloud tenants. We also describe how these challenges were addressed by our prototype implementation, which is called Simple Cache for Cloud (SC2). We demonstrate the effectiveness of these techniques by experimentally evaluating our prototype on a synthetic multitenant workload.

Introduction

Modern websites tend to be dynamic and rich in content, often providing news updates, personalized information, and other data that are invariably stored in databases. When a website becomes popular, the database may be unable to sustain the resulting load, and high response latency for users and other performance challenges ensue. Reducing the end-to-end latency to one's website not only improves customer satisfaction but may also significantly affect one's business. Google, for instance, reported that an extra 0.5 seconds to generate search results lowered traffic rates by 20%, and Amazon found that every 100 ms of latency reduced sales by 1% [1]. Thus, the database emerges as a scalability bottleneck.

Database queries issued on a typical website frequently request the same unmodified information or computation, such as the news stories, content boxes, and template needed to render the front page. This redundancy inspired the development of *memory data caches* in which web servers first check if the result of a query already resides in the memory cache; otherwise, the query is sent to the database, and the result is cached. The common case when generating web content now becomes quick memory access rather than the orders-of-magnitude slower disk access and computational overhead on the database. Memory data caches dramatically improve user-experienced latency and reduce the load on the database.

While similar approaches such as query result caches have historically been an integral part of database services, including IBM DB2*, MySQL**, and Oracle [2, 3], memory data caches have recently become key stand-alone components that reside on a separate cluster in the data center. This decoupling reduces overhead associated with the database and allows administrators to scale out the service simply by adding more servers. For example, *memcached* is a popular in-memory key–value store partitioned across a collection of servers according to the hash of the key [4]. Clients track the map of hash ranges to servers as servers are "oblivious" to one another; that is, the servers do not have information on one another. Sites such as Facebook**, LiveJournal**, YouTube**, and Wikipedia** experience extremely high web request rates and rely on distributed memory data caching services to reduce end-to-end latency for clients and the burden on the database [4]. In 2008, Facebook was reported to use more than 800 specially optimized memcached servers with a total of 28 TB of memory to serve more than 200,000 requests

Digital Object Identifier: 10.1147/JRD.2011.2171649

0018-8646/11/$5.00 © 2011 IBM

per second per server [5]. While this example is extreme, memcached is also a popular service among smaller websites [6].

The cloud continues to attract new customers, particularly due to the appealing lure of scalability and pay-per-use billing model, which together allows the customer's website to seamlessly grow without expensive investment in infrastructure. Businesses using low-level cloud infrastructure as a service (such as Amazon's Elastic Compute Cloud (EC2**) [7]) or a higher level cloud platform as a service (such as Google's App Engine [8]) still need memory caches to reduce latency for their clients and for performance reasons, to such a great extent that Amazon explicitly lists caching as a primary application of its high-memory virtual machines [7], and Google offers a shared memcached service [9]. Whereas caches in other domains are typically of a fixed size, such as hardware caches or web caches, the scalable resource model of the cloud gives rise to the challenge of determining the ideal cache size for the cloud customer to obtain maximum benefit (reduced end-to-end latency and database load) at the lowest cost for resource use.

Multitenant cache

We argue that a *shared* memory cache service is advisable for cloud providers and customers alike. Many cloud providers [8] do not expose the memory infrastructure to their customers, which would allow those customers to manage their own memory data cache service independently. By offering a shared cache service to cloud *tenants*, these providers may improve service quality for their customers and may be able to charge a premium price for such a service. Since cache resources (specifically RAM) are limited and because tenants' workloads tend to be heterogeneous (e.g., due to diurnal variations), a cost-effective strategy for the cloud provider involves dynamic partitioning of cache space among tenants according to the desired service quality.

Even in an infrastructure-as-a-service offering such as EC2, customers may prefer a shared cache service to managing their own cache service on a cluster of virtual machines (VMs). First, when adjusting the total memory size used by the cache, the increments (and decrements) directly relate to the memory provided by the smallest VM offered by the cloud provider, which means that provisioning happens typically only at the coarse granularity of more than 1 GB. Second, the amount of space should vary depending on the workload. For example, a stand-alone cache should be provisioned for peak use. This involves online profiling of resource use over time, which is not supported by current cache services [4, 9]. Third, the automatic allocation or release of resources in response to demand (*elasticity*) in a distributed setting is lacking in current services [4, 9]. Reconfiguration of a memcached cluster is a manual process. In contrast, a shared service can partition space dynamically to give space to those customers who want it and most need it. It can provide elastic resource allocation for tenants and at a much finer granularity than that achieved by using a set of VMs. In addition, it can provide a profile of the customer's cache use to aid with provisioning.

However, a shared memory cache service is associated with a host of challenges. In particular, a simple implementation of a multitenant cache may provide unacceptable service levels for two reasons. On the one hand, heavy contention can cause each tenant to receive an insufficient share of the cache space to store its data, resulting in fewer cache hits and thus higher latency for the tenant's users. On the other hand, a shared memory cache with "tenant-oblivious" policies (such as evicting the *least frequently used* (LFU) or *least recently used* (LRU) entries) favors allocating space to the tenants with higher request rates, thus degrading the service quality in terms of cache hits for the lower rate customers. In the extreme case, the request rate associated with the large online book retail store may simply dwarf the modest yet cacheable workload of a small startup company.

Contributions

These challenges provoke a set of questions. First, what kind of service quality guarantees should be provided by a shared data cache? Second, how can they be enforced by the cloud provider and verified by the customers? Finally, what kind of service and billing model is appropriate for a cloud cache service, given the special properties of a cache compared to, for example, a storage or web server? In answering these questions, we remember that a cloud cache implementation also needs to be simple, efficient, scalable, and robust with respect to failures.

In this paper, we present a new cloud-based caching service called *Simple Cache for Cloud* (SC2), which addresses these challenges. We introduce a service and billing model that takes into consideration the tradeoff between the customer benefit and the cost to the provider. The cloud tenant primarily benefits from a cache due to cache *hits* because they reduce the end-to-end latency for website users, yet cache hits cannot be directly expressed in terms of the primitive resources the provider charges for, which include physical memory, central processing unit (CPU), and network bandwidth. To reconcile the discrepancy between the goals of the customer and the provider, we propose a service model that allows the user to maximize the benefit of cache resources with varying degree of control and automation, whereas the provider charges for physical memory space, as well as network bandwidth and the number of cache requests.

In the simplest case, the user directly controls the allocated cache space and is able to monitor the utility that it gives in terms of cache hits. A more automated alternative

involves a quality of service (QoS) guarantee by the provider, in which users are allocated sufficient cache space (up to a limit) to attain a minimum cache hit rate or other guarantees. In this case, a customer with a cacheable workload will require less cache space, whereas a noncacheable workload may reach the upper limit that the customer is willing to pay for.

Both of these approaches simultaneously exploit the ability to efficiently *profile* cache use at different cache sizes. An implicit assumption here is that the recent cache utilization is indicative of requests made in the near future. This profile, in turn, allows us to predict not just how much benefit (in terms of cache hit rate) the user obtains with her current cache allocation but also what the benefit would be had the cache space been smaller or larger. Such knowledge gives cloud customers visibility into the cost and benefit of adjusting their cache size and allows them to adjust the space use to their needs.

In summary, this paper makes the following contributions.

- We devise a new service model for a cloud caching service in which customers can express their service requirements over time, such as their desired QoS explicit occupancy bounds, or both.
- By viewing memory cache as a cloud utility, we propose a readily understood billing scheme, allowing the users to dynamically gauge the costs and benefits of their current space usage, and adjust the expenditures accordingly.
- We detail the design of SC2, a multitenant distributed cloud caching service. SC2 optimizes the global use of cache resources while simultaneously guaranteeing minimum service quality for all users according to their stated requirements. SC2 also meters cache utilization and exposes this information to customers, as well as their estimated benefit from additional cache space.
- We evaluate an efficient prototype implementation of SC2 on carefully crafted synthetic memory cache workloads.

First, we discuss these points and then provide some concluding remarks.

Billing and service models

The primary resources involved in caching are the memory space for storing cached data, the network bandwidth for transferring data into and out of the cache, and the CPU/network card capacity for handling requests. We charge for all of these resources in our SC2 billing model, but what most distinguishes cache from other cloud services is explicit provisioning of physical memory. Specifically, we charge customers for the cache memory space they use per time unit independently of the benefit they receive from it.

We believe that this model is simple and readily understandable to customers and platform providers alike and that it allows customers a measure of isolation and control of their cost and performance. The approach is also compatible with existing popular cloud application platforms such as Amazon Web Services [7] and Google App Engine [8], which typically present customers with a billing model that charges for use of the underlying hardware resources. We next relate the space use to the benefit received and then provide a service model for the customer that builds on the basic billing scheme.

Measuring space utility

The fundamental tool that we will use to capture the tension between the cost and benefit is the *space utility model*, which expresses the fraction of hits $HR_t(x)$ for a given tenant t and workload over a specific period of time as a function of the cache size x dedicated to the tenant [see **Figure 1(a)**]. Given such a space utility model, each customer may decide what performance level he or she wishes to pay for. For example, most of the benefit afforded by a cache may be achieved at a particular cache size, beyond which there are diminishing returns where cost does not justify the benefits.

SC2 monitors the tenants' workloads, and it continuously computes the space utility model for each tenant based on its current workload. The monitoring mechanism requires no additional actions or effort on the part of customers, and the model is always up to date with the attributes of the current workload. A notable aspect of this mechanism is that it is capable of accurately predicting performance for a range of cache sizes, including numbers *above* the amount of space currently reserved for the tenant. We discuss the mechanism for accomplishing this as part of the SC2 implementation.

Service model

As a basic service model, SC2 provides customers with a tool to directly monitor the space utility model to assist them with reserving the appropriate amount of cache space. Specifically, they can manually adjust their resource consumption according to their available budget and the desired benefits. By default, customers are allocated a fixed amount of cache space that they specify. However, if, for example, nights and weekends are slower for business, then the cache may contain fewer recently used ("hot") items during those periods and less space may be required.

Although the basic model allows cache costs to be readily understood and adjusted, manually manipulating one's cache space is laborious. SC2 provides a more automated framework to manipulate cache space allocation that makes use of the space utility model to provide a variety of QoS guarantees. In addition to explicit occupancy bounds, our system currently supports the following service goals.

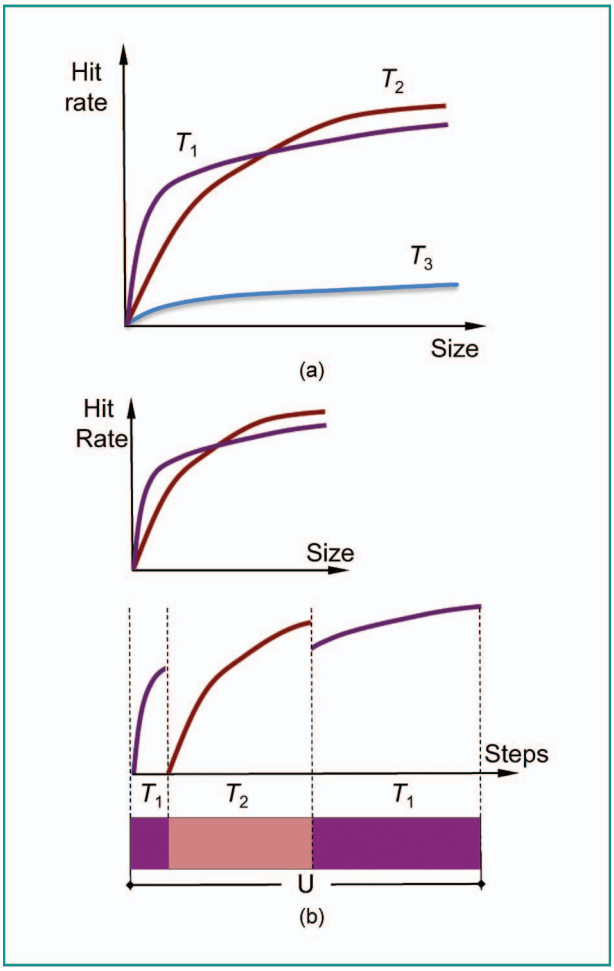

Figure 1

Examples of cache space utility models and their use for determining the target cache allocation. (a) Space utility models for the workloads of three tenants T_1, T_2, and T_3. The workloads of tenants T_1 and T_2 are more cacheable than that of tenant T_3. (b) Sequence of the space allocation decisions taken at each step of the greedy algorithm when it is used to partition the space of size U between tenants T_1 and T_2. The resulting partition will maximize the total hit rate for the portion of the cache occupied by these two tenants.

Minimum target hit rate

Customers are allowed to set hit rate goals and couple them with maximum space bounds. Formally, minimum cache space x is allocated to tenant t such that $HR_t(x) \geq h_t$ subject to $l_t \leq x \leq u_t$, where h_t, l_t, and u_t are set by the tenant. Given such a specification, the system dynamically adapts the space reserved for the tenant to be the least amount of space that achieves the goal of *minimum target hit rate* based on the tenant's computed space utility model. Tenants provide minimum and maximum space bounds l_t and u_t so that they may limit their costs while ensuring minimum space allocation in case the workload changes unexpectedly, making the target hit rate too expensive (or impossible) to achieve. Under this guarantee, tenants pay for the actual space used per time unit rather than for a fixed reserved amount.

Performance gain threshold

A second QoS goal allows tenants to indicate a *threshold of performance gain* they are willing to pay for, rather than an explicit hit rate. The system will assign available space to a tenant if each additional unit of space will increase the tenant's hit rate by $inc_t\%$ (where inc_t is the threshold set by the customer). Here, as well, customers can set lower and upper bounds l_t and u_t on their space consumption. This QoS goal is equivalent to increasing x while $HR_t(x) \geq HR_t(x-1) \times (100 + inc_t)\%$ subject to $l_t \leq x \leq u_t$. Since the space utility curves are concave in practice [10, 11], there is a unique maximum point x that satisfies the constraint.

Maximize hit rate for tenant groups

The final guarantee generalizes the minimum target hit rate for *tenant groups* to exploit heterogeneous workloads. Let T denote a group of tenants. The customer can reserve some total amount of space S_T for T and pay accordingly. Our space utility profiling mechanism then automatically optimizes performance *within the group* by giving space to those tenants that most benefit from it. Formally, if r_t denotes the average request rate of tenant t, we maximize $\sum_{t \in T} r_t \cdot HR_t(x_t)$ subject to $\sum_{t \in T} x_t = S_T$. Note that there is tension here between performance and isolation since the amount of cache space allocated to a tenant now depends on the behavior of the other tenants. To give an example of a tenant group, consider a company with three departments that all access SC2. Each department is given a tenant identifier T_1, T_2, and T_3, and the space utility metric for each tenant is shown in Figure 1(a). Under this guarantee, the system would dynamically allocate more space to T_1 and T_2 than T_3 because of their high benefit from the cache. These tenants could also represent the components of a composite workload, such as the database index and data entries within the same application, each of which has different cache properties. As before, minimum cache space l_t for each tenant t within a group can be specified to allow tenants to have sufficient space for profiling their cache use.

Provisioning schedules

Workloads can change over time; thus, a provisioning decision that is optimal at peak hours, for instance, may actually be overprovisioned at other times. To accommodate such predictable fluctuations in workloads, customers specify their needs in a *provisioning schedule*. Each entry in the provisioning schedule has two parts. The first is a *provisioning specification*, which is one of the QoS goals

aforementioned or an explicit occupancy bound. The second is a time interval during which this QoS specification is in effect. For example, the schedule entry (`Minimum target hit rate` $h_t = 80\%$ `with` $l_t =$ `10 MB and` $u_t =$ `1 GB`, {`Mon − Fri 00 : 00 − 23 : 59`) means that tenant t wishes to maintain an 80% hit rate during working days so long as the cache space needed is between 10 MB and 1 GB. An entry in the provisioning schedule is defined for a single tenant, such as in the example, or a group of tenants when the goal is to maximize hit rate for that group. The syntax and semantics for time periods are similar to those used in UNIX** cron jobs, including the ability to specify recurring time periods such as certain times of certain days. The time periods within a single provisioning schedule should not overlap, and at most, one provisioning specification should apply to a tenant at any point in time.

In the ongoing work, we generalize the conditions at which rules become effective beyond time-based triggers. For example, provisioning specifications can be activated in response to certain request-rate thresholds.

Service application programming interface

Users can interact with SC2 via a *cache access application programming interface (API)* to get and set cache keys and a *management* API to set provisioning schedules and retrieve cache statistics.

The cache access API provides an interface commonly used in key–value stores and memcached [4]. It includes operations to retrieve the cached object associated with a given *key* (get(key)), add a new key–object pair to the cache (add(key, object)), modify the value of an existing cached object (set(key, object)), and delete the object for a given key (delete(key)). In addition, each object is associated with an expiration time, and the objects that have not been accessed for at least the expiration time duration are purged from the cache. To simplify the presentation, we will assume that all objects are of a fixed size corresponding to a single unit of space. We note that this does not affect the generality because larger objects may be divided into chunks by an intermediate layer, and smaller objects may be similarly consolidated.

The management API includes methods for setting and modifying the tenant provisioning schedule, monitoring the current cache usage profile, and querying the billing information.

Implementation overview

Figure 2 shows the architecture of SC2. Cache objects are partitioned across a collection of caching servers. The partitioning and lookup mechanisms use a consistent hashing scheme, which ensures that both total and per-tenant storage loads are evenly spread across the servers [12, 13]. The current server membership and consistent hashing metadata are maintained by the tenant deployment and server presence component, which, in our prototype, is supported by ZooKeeper [14].

The centralized space utility profiler monitors the space usage of tenants and interacts with the hit stats monitor module on each server to collect the local usage statistics. The local statistics are used to approximate the global space utility model HR_t for each tenant t [see Figure 1(a)]. The space utility model is used by the partition optimizer to decide how to partition the global caching storage among tenants in order to satisfy the QoS specification and constraints imposed by the provisioning schedule. The resulting partition is passed to the provisioner, which calculates the target space occupancy for each tenant on each server and communicates this information to all servers in the system. Furthermore, the space utility model is exposed to the cloud customers, as previously detailed.

If the total available memory on the servers is not sufficient to satisfy the constraints of the provisioning schedule, the provisioner may instantiate a new cache server out of the pool of available virtualized computing resources. Later, the provisioner may decide to consolidate the cache space in fewer servers and shutdown the surplus servers. Whenever the number of servers changes, the provisioner updates the consistent hashing metadata, and it may initiate the transfer of objects between servers to accommodate the range split or merge.

The replacement policy component on each server uses the target occupancies to make allocation and replacement decisions in the local caches. Our replacement policy implementation is based on the well-known clock algorithm [15], which was augmented with an occupancy enforcement mechanism to ensure that the tenant occupancies converge to and stay within the limits imposed by the provisioner.

Implementation highlights

We now review the design and implementation of the most interesting components in our design in more detail. These are the modified clock replacement policy for enforcing a tenant cache space partition, a per-node hit stats monitor, and a central space utility profiler to collect global statistics about the utility of cache for individual tenants and the partition optimizer and provisioner, which determine an updated tenant space allocation on each node based on the space utility profile and tenant provisioning schedule. These components are shown in blue in Figure 2.

Predictive space utility profiling

There are two challenges in computing a space utility model for tenants. First, even in retrospect, it is difficult to know what hit rate would have been achieved with a cache size different from what the tenant was allocated, unless all cache operations have been recorded. Accordingly, we will

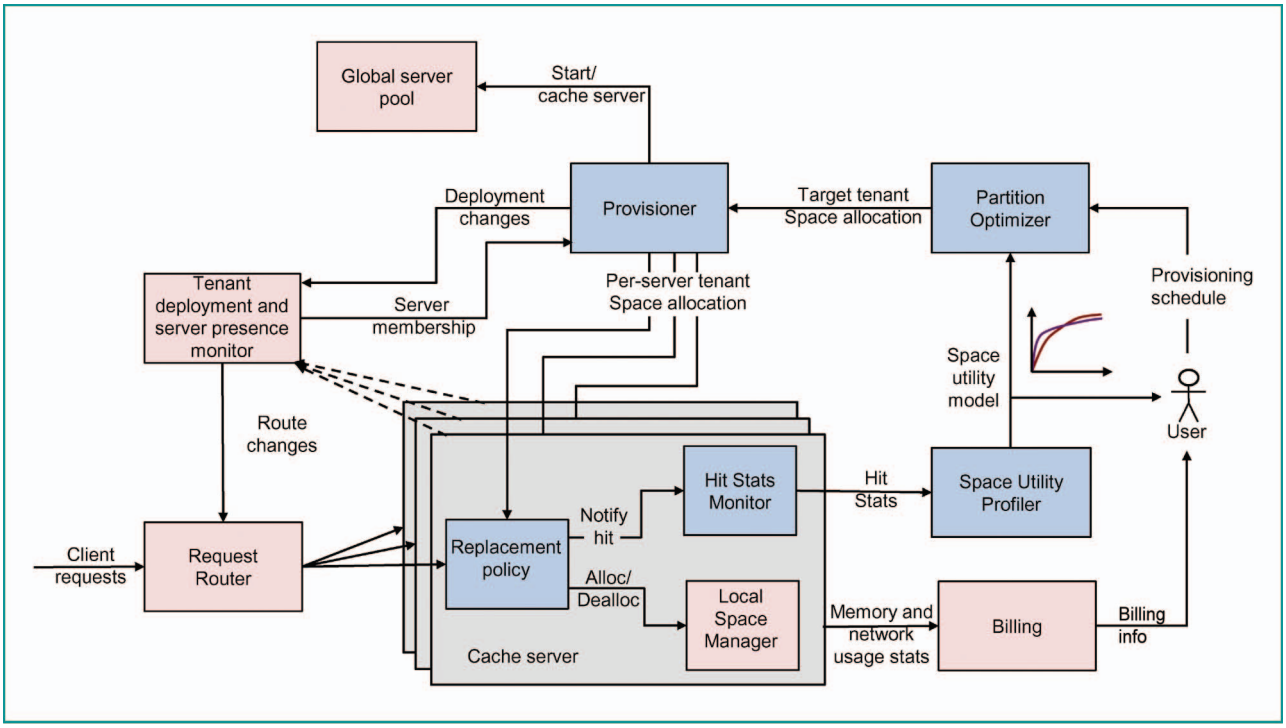

Figure 2

Abstraction of the SC2 system architecture. Cache requests arrive at a request router, which forwards them to the appropriate cache server to access, update, or remove an element from the cache. If the local space manager is unable to allocate ("Alloc" in the figure) space for a new object, the replacement policy will evict an element from some tenant such that the memory partition corresponds to the desired tenant space allocation, and then try again. Cache hits and misses for each tenant are recorded by the hit stats monitor, and they are periodically submitted to a global space utility profiler, which derives a space utility model (e.g., in the form of curve plots) of the aggregate cache space of tenant. The tenant is given access to these curves, as well as billing information corresponding to network and memory use, to help define a provisioning schedule of cache use. The partition optimizer uses the provisioning schedule and space utility models to create a target space partition for tenants, which, in turn, is divided for the cache servers by the provisioner. Finally, the provisioner and the tenant deployment and server presence monitor manage the cache servers, monitoring them and restarting or adding new VMs as needed, and notifying the request router on routes changes.

show how SC2 estimates the answer using profiling with minimal overhead. The second issue is that hit rates dynamically change as workloads shift. We assume that the past predicts the near future, and thus, let $HR_t(x)$ denote the fraction of hits that tenant t has received over a specific period of time (a parameter) assuming that the cache size of t has been x, and use this rate to predict the proportion of hits that t will receive in the near future at the same cache size.

The goal of the space utility profiling mechanism in Figure 1(a) is to automatically estimate the space utility model for each tenant. The tenant space utility models are used to drive the space allocation decisions by the partition optimizer and are exposed to the customer via the management API. Below, we first describe the methodology underlying our implementation of the space utility profiling, and then explain how it was adapted to a distributed setting.

Replacement policy and profiling methodology

It is well known that the space utility curve can be extracted from the LRU replacement policy [16]. The LRU stack for a cache of size C_1 is the prefix of the LRU stack for any cache size $C_2 \geq C_1$. Hence, in order to obtain the hit rate for all cache sizes between 1 and C, it is sufficient to maintain the histogram of hits for each *stack distance* (i.e., the distance from the head of the LRU stack) that has been observed in the run. The space utility model is then the cumulative distribution function of this stack distance hit histogram. Unfortunately, there is a subtle issue here that precludes an efficient implementation. LRU is typically implemented using linked lists for efficient reordering on each access, but this means that, determining the exact stack distance of an entry requires traversal or enumeration of the list on every hit. More sophisticated schemes for estimating stack distances in LRU that trade off efficiency and accuracy have been recently proposed [17].

Moreover, LRU has another serious problem. On every cache hit, the entry being accessed must be moved to the top of the stack, which is an expensive operation in multithreaded environments and causes contention on the list head as hits are serialized behind a single global lock [18, 19]. Many production systems such as DB2, BSD UNIX, IBM AIX*, Apache Derby, and PostgreSQL [18–20] have instead used the clock replacement algorithm [15] that has minimal overhead on cache hits. We made the same design choice in SC2.

In clock, the entries in the cache are organized into a circular array, which is traversed using a pointer called the *hand*. Each entry is associated with a *recently used* bit, which is set every time the entry is accessed (by a get or a set request). Whenever a client issues a request to add a new (key, object) pair to the cache and there is an insufficient free storage space available to accommodate the object, the entries are traversed in a clockwise fashion starting from the one currently pointed to by the hand until an entry, whose recently used bit is unset, is found. The object associated with this entry is then replaced with the new object.

Since stack distances are not explicitly available with clock, we devised the following scheme to estimate them. An entry e in the clock is called *active* if the recently used bit associated with e is equal to 1 and *idle* if it is 0. For a tenant t, let na_t and ni_t be the current number of the active and idle clock entries, and let n_t be the current total number of entries owned by t. The intuition behind our approach is that a hit on an active entry in our clock is as if an entry at stack distance between 1 and na_t in the corresponding LRU stack was hit since those are the most recently used entries [21]. However, we do not know which stack distance exactly was hit; hence, we place equal probability $1/na_t$ at each intermediate stack distance. Similarly, an access to an idle entry would correspond to a hit to an entry of stack distance between $na_t + 1$ and n_t in the LRU stack. We then add the probabilities on each stack distance to create an expected stack distance histogram.

More formally, the stack distance hit histogram is derived from two arrays called *active* and *idle*, where $active_t[k]$ accumulates the number of hits that have occurred at stack distances between 1 and k, and $idle_t[k]$ counts the hits that occur at stack distances between $k + 1$ and n_t. Specifically, whenever an active entry owned by tenant t is accessed, we add $1/na_t$ to the value stored in $active[na_t]$, and for an idle entry, we add $1/ni_t$ to $idle[na_t + 1]$. Note that other probability distributions can be easily supported. Stack distance hit histogram H_t is given by the following expression:

$$H_t[k] = \sum_{i=k}^{n_t} active_t[i] + \sum_{i=1}^{k} idle_t[i],$$

for all cache sizes k, $1 \le k \le n_t$.

The space utility model HR_t for tenant t is thus obtained as follows:

$$HR_t[k] = \frac{1}{r_t} \sum_{i=1}^{k} H_t[i], \text{ for all cache sizes } k, 1 \le k \le n_t.$$

Here, r_t is the total number of cache requests that have been received by t. In order to reduce the space for storing the profiling metadata, the hit statistics (i.e., $active_t$ and $idle_t$), as well as model M_t, are maintained at a granularity of coarse-grained allocation blocks of a configured size, and not for each individual stack distance. To ensure that the models stay current in the face of dynamic changes in the tenant workloads, the accumulated statistics are periodically aged by subjecting the values stored in r_t, $active_t$, and $idle_t$ to an exponential average, and the space utility model is recomputed.

To estimate each tenant's future demand for the caching space based on the past history, the hit statistics are also maintained for a subset of the keys associated with the objects that have been recently evicted from the cache [22]. In particular, whenever an entry is selected for eviction by the replacement policy, we only evict the associated object while retaining the metadata for the entry itself and the object key. The number of additional objectless, i.e., "*ghost*," entries being maintained for each tenant is specified as a fraction of the tenant's current occupancy and was set to 10% in our experiments. The target ghost occupancies are enforced through the same clock-based mechanism as the one described above.

We note that the quality of the space utility estimate and replacement decisions relies on a reasonable rate of accesses relative to the cache size. However, some users such as Facebook report to effectively use their caching tier as long-term memory storage with a very large hit rate. Our approach would need to be modified to support such scenarios in which the rate of evictions is too low for clock to toggle the recently used bits and deliver insightful estimates of valuable entries in the cache. An extension we are investigating involves introducing a background thread to age clock entries after a certain interval.

Distributed implementation

We now briefly describe how the above techniques are adapted to a distributed setting. For brevity, we only present the basic technique and omit details on handling failures, message delays, and loss.

The hit statistics collection is orchestrated by the space utility profiler and proceeds in *rounds*. The rounds are periodically initiated (with a long period) by the space utility profiler by broadcasting a unique round number to all of the caching servers. The hit stats monitor module on node i responds by periodically (with a short period) reporting for each tenant t the number of active entries $na_{t,i}$ and the number of hits $ha_{t,i}$ that have occurred on those active entries

since the previous report and, similarly, $ni_{t,i}$ and $hi_{t,i}$ for idle entries. Since the number of active and idle entries may vary between reports, the server communicates the weighted average

$$(na_{t,i}, ha_{t,i}) = \left(\left(\sum_{j=1}^{J} na_{t,i}(j) \cdot ha_{t,i}(j) \Big/ \sum_{j=1}^{J} ha_{t,i}(j) \right), \sum_{j=1}^{J} ha_{t,i}(j) \right)$$

of the J hit statistics $(na_{t,i}(j), ha_{t,i}(j))$ for $j = 1, \ldots, J$, which server i recorded in the last period. The same is done for idle entries. The balance between the accuracy of the resulting cache utility model and the overhead due to the traffic reports is thus controlled by the report frequency. Each statistics report message is tagged with a pair (rn, sn), where rn is the current round number, and sn is the serial report number within round rn.

Given a collection of statistics reports $\{(na_{t,1}, ni_{t,1}, ha_{t,1}, hi_{t,1}), \ldots, (na_{t,N}, ni_{t,N}, ha_{t,N}, hi_{t,N})\}$ for tenant t tagged with the same (rn, sn), the space utility profiler updates the $active_t$ and $idle_t$ arrays in the following fashion.

Let $active_t[k] = active_t[k] + \sum_{i=1}^{N} ha_{t,i}/k$ for $k = \sum_{i=1}^{N} na_{t,i}$, and let $idle_t[k] = idle_t[k] + \sum_{i=1}^{N} hi_{t,i}/k$ for $k = \sum_{i=1}^{N} ni_{t,i}$, as long as $1 \le k \le n_t$.

The *active* and *idle* arrays are periodically aged using exponential averaging, and the space utility model for each tenant is regenerated as explained above.

Partition optimizer

The partition optimizer is responsible for deciding how to partition the total available caching space among the tenants based on the constraints imposed by the provisioning schedule and the tenants' space utility models HR_t. Partitioning is performed differently depending on the provisioning specification in the provisioning schedule. The output set of allocation constraints is passed on to the provisioner. Note that explicit occupancy limits do not require any special treatment as they are simply merged into the set of the output allocation constraints.

Minimum target hit rate h_t using at most u_t space

If there exists s such that $HR_t[s] = h_t$, set the minimum occupancy limit l_t for t to $\min(s, u_t)$. Otherwise, set $l_t = u_t$. Add the range $[l_t, u_t]$ to the set of the output allocation constraints.

Performance gain threshold $inc_t\%$ using at most u_t space

We increase the space allocation s_t to tenant t as long as the relative improvement in the hit rate of t is at least $inc_t\%$. More formally, if there exists a Δ such that $HR_t[s_t + \Delta]/HR_t[s_t] \ge 1 + inc_t/100$, set the minimum occupancy limit of t (l_t) to $\min(s_t + \Delta, u_t)$ for the highest such Δ. Otherwise, set $l_t = s_t$. Add l_t to the set of the output allocation constraints.

Maximize hit rate for a tenant group T using at most U_T space in total

We calculate the minimum occupancy limit l_t for each tenant $t \in T$ using the following greedy partitioning algorithm similar to [10, 23] [see **Figure 1(b)**]:

```
Initialize l_t = 0 for each t ∈ T;
for k = 1 to U_T do:
  Find tenant t that maximizes r_t(HR_t[l_t + 1] − HR_t[l_t]);
  Set l_t = l_t + 1;
done
```

Add U_T and l_t for each tenant t to the set of the output allocation constraints. This algorithm produces a partition with maximum total utility when utility curves HR_t are concave, as is generally true in practice [10, 11].

Occupancy enforcement in clock

The provisioner refines the cache space partition computed by the partition optimizer to produce tenant occupancy constraints for each cache server (see Figure 2). In order to satisfy these occupancy constraints on the cache server, we augmented the basic clock replacement mechanism as follows. First, to conserve space and simplify concurrency control, all cache entries are maintained within a single circular array and traversed by the same hand pointer irrespective of what tenant owns them. Next, whenever tenant t wishes to add a new object *obj* to the cache, we first check if there is sufficient available free space to accommodate *obj* and if growing the allocation of t complies with the current occupancy limits. If so, we instruct the local space manager to allocate storage for *obj*, and the corresponding entry is added to the clock (with the recently used bit set to 1). Otherwise, the array is traversed as in the basic clock replacement scheme until we find an entry e owned by tenant t' such that both of the following conditions are satisfied: 1) the recently used bit of e is 0, and 2) transferring one object from tenant t' to t results in an allocation that is not worse than the current one relative to the current occupancy limits. The object associated with entry e is then replaced with *obj*.

More formally, consider the case when the tenant allocation constraints for each tenant t are specified as ranges $[l_t, u_t]$ of lower and upper occupancy bounds. Then, an object owned by tenant t' will be replaced with an object owned by t if their current occupancies s_t and $s_{t'}$ satisfy either one of the following two conditions: 1) $s_t < u_t$ and $s_{t'} > l_{t'}$, or 2) $s_t \ge l_t$ and $t = t'$. Intuition suggests making progress toward satisfying the boundary conditions for all tenants, and when they are satisfied, trying to evict the first possible entry encountered without violating the conditions.

Evaluation

We now turn our attention to assessing the effectiveness of our prototype implementation of SC2. Many of the

features in SC2 are either qualitative (such as provisioning schedules) or well known in the literature (such as the space manager). Accordingly, the focus of our evaluation will be on the *novel* attributes in SC2 that distinguish it from other systems and can be meaningfully measured. In particular, we will experiment with the profiling, partitioning, and replacement mechanisms in SC2, as shown in blue in Figure 2. The high-level question we intend to answer with our experiments is "How effective is the SC2 space allocation mechanism?" For the sake of brevity, we focus on the most challenging scenario we can undertake to address this question, which is to dynamically maximize the total hit rate for a group of tenants in the system.

Workloads

Appropriate input is the key to an insightful experimental evaluation of a system. In our case, the input workload is a series of *get, add, set*, and *delete* operations on object keys and the identifier of the tenant that issued the request. Because we are not aware of publically available request traces to multitenant caches, we carefully crafted synthetic workloads to give a controlled evaluation of our algorithms. For simplicity, we focus on a cache, which cannot fit all objects in memory, and issue only *get* and *set* operations. We defined the ratio of gets to sets to be 4 to 1.

Our workload generator is able to synthesize a workload for a multitenant cache that simultaneously produces the target object *access frequency distribution* and the target *temporal locality distribution*. Here, temporal locality defines the number of distinct requests that were made between accesses to the same (nonspecific) cache object, and it is analogous to the LRU stack distance described earlier. Note that these distributions are not independent from one another; for instance, the requests to an item with high access frequency tend to be temporally close, which makes the ability to control both distributions at the same time both nontrivial and useful.

A simple scenario to test our partitioning mechanism is to compare the space allocated to tenants that are polluting the cache with entries that will not be reused to the entries of tenants that indeed benefit from adequate cache space. We define a *cacheable* tenant as one who frequently accesses recently accessed objects. More specifically, given that the tenant has 500 distinct objects, we define the temporal locality distribution to select items with an LRU stack distance of less than 250 with at least 99% probability. In contrast, a *noncacheable* tenant accesses a large number of objects (we set this arbitrarily to 5,750 in our experiment) and tends to choose old items; for example, if the memory cache is used to store each result during the scan of a database table. In our workload, the noncacheable tenant will not request an item again unless it has observed over 4,000 distinct ones (again with >99% probability). For both types of tenants, we use the uniform random distribution to choose among equivalent items and to define the object access frequency distribution.

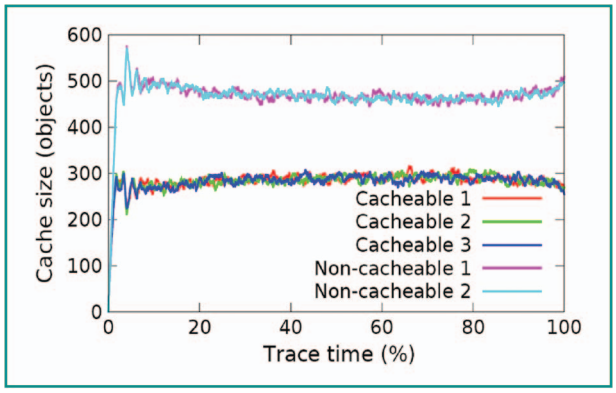

Figure 3

Experiment showing a "tenant-oblivious" clock replacement policy allocating space to three cacheable tenants and two noncacheable tenants using a synthetic trace of 150,000 requests. The cache can fit at most 1,800 objects, and the cacheable tenants access at most 500 objects. The noncacheable tenants finally receive a higher portion of the cache because they evict other cache entries more aggressively than the cacheable ones.

Experimental results

The core of the cache server implementation in our Java** prototype of SC2 is based on a multithreaded clock code from the Apache Derby database implementation [24], augmented with support for multiple tenants such as space partitioning, profiling, occupancy enforcement, and so forth. In our experiments, we will use the original clock without any multitenancy support as the baseline for comparison.

For our experiment, we subject an SC2 node to a workload consisting of requests from three cacheable tenants and two noncacheable ones, and we plot the tenant space allocation over time at every 20 requests. Every tenant issues requests at an equal rate, and all objects are of equal size. The cache can hold 1,800 objects, which means that the three cacheable tenants could easily fit each of their 500 objects in the cache memory if the space were manually partitioned. However, the partitioning mechanism is dynamic and adaptive, and it should automatically give less space to noncacheable tenants. We run the partition optimizer every 1,250 requests and set the ghost lists to only 10% of the entire clock or 36 entries per tenant.

In the experimental results in **Figure 3**, it is shown that the "tenant-oblivious" clock allocates more space to *noncacheable* tenants (1 and 2) than the cacheable ones (1, 2, and 3). In other words, the replacement policy is performing even worse than what intuition might suggest, which is that, since all tenants issue requests at the same rate, they should be given an equal share of cache space. The

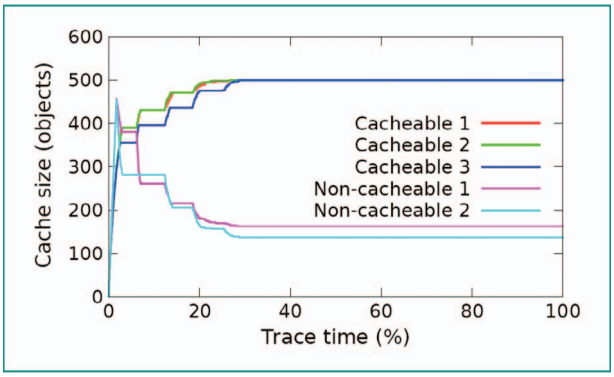

Figure 4

Experiment showing the sequence of space allocations made by SC2 to three cacheable tenants and two noncacheable tenants in a synthetic trace of 150,000 requests. The cache can fit at most 1,800 objects, and the partition optimizer is run every 1,250 requests. The cacheable tenants request at most 500 different objects; thus, the space allocation in the graph converges to an optimal one (100% hit rate for the cacheable tenants).

explanation is that the noncacheable tenants are requesting items that are *not* in the cache, thus producing many evictions from the cache and increasing space use for that tenant. The rate of the evictions by noncacheable tenants exceeds that of the cacheable tenants, which tend to request entries that are already cached, and this translates to relatively less cache space being allocated to cacheable tenants.

If we now look at the results for SC2 in **Figure 4**, we see that tenant space partition converges to become nearly optimal. The cacheable tenants are each allocated around 500 objects, which is sufficient for them to store their entire set of keys, and the noncacheable tenants share the remaining space. The speed of convergence is directly related to the number of requests needed to reach the new target size for the tenant and the ghost list size. The intervals between the jumps in the graph show the partition optimizer at work, and the jump size is at most the size of the ghost list.

The hit rate in SC2 for all tenants combined for the duration of the trace was 65%, whereas for the tenant-oblivious cache, it was 44%. Overall, SC2 allocates space to those tenants that most benefit from the space, and after partitioning converged, the hit rate for cacheable tenants is 100%.

Related work

Memory caches are gaining acceptance within industry clouds [3–9, 25]. This development parallels the so-called NoSQL trend in the database community, which rejects the traditional relational model in favor of key–value stores, and sacrifices strong consistency for horizontal scalability [20, 26]. Currently, key–value stores are becoming the quintessential scalable cloud databases. The unique contribution of SC2 in this area involves that multitenancy aspects such as sharing and service quality of such data stores, which have not been previously explored in detail in the literature to the best of our knowledge.

The variants of the cache partitioning problem that we discussed arise in various other contexts such as hardware caches [21], web caches [11], storage caches [27–30], and in database cache management [20, 31]. These papers are focused on fair service differentiation and QoS goals [11, 20, 28], maximizing the total cache hit rate [21, 31], or a combination of the two as in SC2 [27, 29]. Ko et al. [28] and Storm et al. [20] discuss control theoretic approaches to stabilizing cache partitioning over time, which we intend to incorporate into a future version of SC2. However, none of the work considers profiling and space partitioning in the context of the clock replacement policy. The authors provide only basic schemes for profiling and ignore distributed profiling, and the algorithms are unable to estimate (without significant modifications) the stack distances at various cache sizes as exposed by the space utility model of SC2.

Conclusion

In this paper, we have proposed a new cloud-based caching service called SC2, and we have addressed various challenges associated with its design and implementation. SC2 supports a simple service model allowing the users to express their requirements as a combination of quality-of-service guarantees and explicit occupancy bounds. The system exposes a readily understood billing model allowing the users to dynamically gauge the costs and benefits of their current space usage and adjust the expenditures accordingly. Our prototype implementation of SC2 introduces new techniques for efficient profiling and partitioning of the cache space among multiple service tenants and validates their effectiveness through an experimental study. Our ongoing work is focusing on extending our current prototype into a fully fledged cloud service, which, among other things, includes supporting high availability, replication, and using solid-state drives for secondary storage.

References

1. *Latency is Everywhere and It Costs You Sales—How to Crush It.* [Online]. Available: http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it
2. Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier database

caching for e-business," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, 2002, pp. 600–611.
3. *CSQL Main Memory Database Cache*. [Online]. Available: http://databasecache.com
4. B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 124, Aug. 2004.
5. P. Saab, *Scaling Memcached at Facebook*. [Online]. Available: http://www.facebook.com/note.php?note_id=39391378919
6. P. Galbraith, *Developing Web Applications With Apache, Mysql, Memcached, and Perl*. Birmingham, U.K.: Wrox Press, 2009.
7. *Amazon Elastic Compute Cloud*. [Online]. Available: http://aws.amazon.com/ec2/
8. D. Sanderson, *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. Sebastopol, CA: O'Reilly Media, 2009.
9. *Google Memcache*. [Online]. Available: http://code.google.com/appengine/docs/python/memcache
10. H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Trans. Comput.*, vol. 41, no. 9, pp. 1054–1068, Sep. 1992.
11. Y. Lu, C. Lu, T. Abdelzaher, and G. Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching services," in *Proc. IEEE Int. Workshop Quality Serv.*, Miami Beach, FL, May 2002, pp. 23–32.
12. 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," in *Proc. ACM Symp. Theory Comput.*, New York, 1997, pp. 654–663.
13. D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Comput. Netw.*, vol. 31, no. 11–16, pp. 1203–1213, May 1999.
14. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. USENIX Annu. Technol. Conf.*, Boston, MA, 2010.
15. F. J. Corbato, "A paging experiment with the multics system," MIT Press, Cambridge, MA, MIT Project MAC Rep. MAC-M-384, May 1968.
16. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.
17. D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, White Plains, NY, Mar. 2010, pp. 55–65.
18. S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. USENIX Conf. File Storage Technol.*, 2004, pp. 187–200.
19. S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proc. USENIX Annu. Technol. Conf.*, Anaheim, CA, Apr. 2005, pp. 323–336.
20. A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, "Adaptive self-tuning memory in DB2," in *Proc. Int. Conf. Very Large Data Bases*, Seoul, Korea, 2006, pp. 1081–1092.
21. K. Kedzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Atlanta, GA, 2010, pp. 1–12.
22. N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. USENIX Conf. File Storage Technol.*, San Francisco, CA, 2003, pp. 115–130.
23. M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, Washington, DC, 2006, pp. 423–432.
24. P. C. Zikopoulos, G. Baklarz, and D. Scott, *Apache Derby—Off to the Races: Includes Details of IBM Cloudscape*. New York: IBM Press, 2010.
25. *Schooner Membrain*. [Online]. Available: http://www.schoonerinfotech.com/products/schooner_membrain
26. M. Stonebraker, "SQL databases v. NoSQL databases," *Commun. ACM*, vol. 53, no. 4, pp. 10–11, Apr. 2010.
27. R. Prabhakar, S. Srikantaiah, C. Patrick, and M. Kandemir, "Dynamic storage cache allocation in multi-server architectures," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, New York, 2009.
28. B.-J. Ko, K.-W. Lee, K. Amiri, and S. Calo, "Scalable service differentiation in a shared storage cache," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, Providence, RI, 2003, pp. 184–193.
29. C. M. Patrick, R. Garg, S. W. Son, and M. Kandemir, "Improving I/O performance using soft-QoS-based dynamic storage cache partitioning," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, 2009, pp. 1–10.
30. L. Chakraborty and A. Singh, "A utility-based approach to cost-aware caching in heterogeneous storage systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10.
31. G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza, "Dynamic resource allocation for database servers running on virtual storage," in *Proc. USENIX Conf. File Storage Technol.*, Feb. 2009, pp. 71–84.

Received February 6, 2011; accepted for publication March 16, 2011.

Gregory Chockler *IBM Research Division, Haifa Research Lab, Haifa University Campus, Mount Carmel 31905, Israel (chockler@il.ibm.com).* Dr. Chockler is a Research Staff Member in the Distributed Middleware Group at IBM Research, Haifa Research Lab. He received his Ph.D. degree in computer science from the Hebrew University of Jerusalem in 2003. He then joined the Theory of Distributed Systems Group at the Computer Science and Artificial Intelligence Lab of MIT as a postdoctoral associate. He has been employed by IBM Research since 2005. Dr. Chockler's research interests span all areas of distributed systems, and in particular, cloud computing, fault tolerance, peer-to-peer computing, and middleware virtualization technologies. In 2010, he received an IBM Outstanding Technical Accomplishment Award for his contribution to middleware virtualization technologies and products. He is an author or coauthor of 4 patents and more than 60 technical papers. He presently serves on the Editorial Board of *Information Processing Letters* and is a guest editor of the special issue on cloud computing for *Journal of Parallel and Distributed Computing* (JPDC). He is a cofounder of the ACM/SIGOPS (Special Interest Group on Operating Systems) Workshop on Large-Scale Distributed Systems and Middleware (LADIS).

Guy Laden *IBM Research Division, Haifa Research Lab, Haifa University Campus, Mount Carmel 31905, Israel (*laden@il.ibm.com*).* Mr. Laden is a Research Staff Member in the Distributed Middleware Group at IBM Research, Haifa Research Lab. Before joining IBM in 2002, he studied computer science at Tel Aviv University and participated in several startup ventures, including Cyota, which was later acquired by RSA Security, Inc. He joined the Storage Systems Research Group at IBM Research, Haifa Research Lab, as a researcher in 2002 and worked on object disks, continuous data protection, and cloud storage. He moved to the Distributed Middleware Group in 2010. He was awarded an IBM Technical Accomplishment Award in 2008 for his contribution to continuous data protection technology and products, and is coauthor of several patents and technical papers.

Ymir Vigfusson *IBM Research Division, Haifa Research Lab, Haifa University Campus, Mount Carmel 31905, Israel, and Reykjavik University, Menntavegur 1, 105 Reykjavik, Iceland (ymirv@il.ibm.com).* Dr. Vigfusson is a Research Staff Member at IBM Research in Haifa and an Assistant Professor at Reykjavik University. He received his Ph.D. degree in computer science from Cornell University in August 2009, where he researched ways to exploit group similarity and improve scalability in distributed systems. His dissertation was nominated by Cornell for the ACM Doctoral Dissertation Award. Dr. Vigfusson's research projects include creating and optimizing systems and algorithms for distributed settings, and ensuring multicast works in a variety of environments. His work has been partially supported by a Fulbright Scholarship and a Yahoo! Research Grant.