Elastic Cloud Caches for Accelerating Service-Oriented Computations

David Chiu† Apeksha Shetty Gagan Agrawal
† School of Engineering and Computer Science, Washington State University, Vancouver
Department of Computer Science and Engineering, The Ohio State University

***Abstract*—Computing as a utility, that is, on-demand access to computing and storage infrastructure, has emerged in the form of the Cloud. In this model of computing, elastic resource allocation, i.e., the ability to scale resource allocation for specific applications, should be optimized to manage cost versus performance. Meanwhile, the wake of the information sharing/mining age is invoking a pervasive sharing of Web services and data sets in the Cloud, and at the same time, many data-intensive scientific applications are being expressed as these services. In this paper, we explore an approach to accelerate service processing in a Cloud setting. We have developed a cooperative scheme for caching data output from services for reuse. We propose algorithms for scaling our cache system up during peak querying times, and back down to save costs. Using the Amazon EC2 public Cloud, a detailed evaluation of our system has been performed, considering speed up and elastic scalability in terms resource allocation and relaxation.**

I. Introduction

The diminishing cost of bandwidth, storage, and processing elements, together with advancements in virtualization technology, have allowed for the subsistence of *computing as a utility*. This utility computing model, the Cloud, ventures to offer users on-demand access to ostensibly unlimited computing and storage infrastructure. This model has proved to be highly desirable for various stakeholders within the industry and the academe, as their localized data centers can now be outsourced to the Cloud to save on such costs as personnel, maintenance, and resource usage [3]. Providers, including (but certainly not limited to) Amazon AWS [1], Microsoft [34], and Google [23] have already made great strides towards ushering Cloud computing to the mainstream.

Particularly, scientific application users have begun harnessing the Cloud's *elastic* properties, i.e., on-demand allocation and relaxation of storage and compute resources [50], [14], [42]. Additionally, such applications have lately embraced the Web service paradigm [12] for processing and communications within distributed computing environments. Among various reasons, the interoperability and sharing/discovery capabilities are chief objectives for their adoption. Indeed, the Globus Toolkit [17] has been employed to support service-oriented science for a number of years [18]. These observations certainly do not elude scientific Cloud applications – indeed, some speculate that Clouds will eventually host a multitude of services, shared by various parties, that can be strung together like building-blocks to generate larger, more meaningful applications in processes known as *service composition*, *mashups*, and *service workflows* [22].

*** This work is dedicated to Yuri Breitbart (1940—2010), Ohio Board of Regents (OBR) Distinguished Professor at Kent State University, who will always be remembered for his wisdom, empathy, and unwavering guidance of his students.

Situations within certain composite service applications often invoke high numbers of requests due to heightened interest from various users. In a recent, real-world example of this so-called *query-intensive* phenomenon, the catastrophic earthquake in Haiti generated massive amounts of concern and activity from the general public. This abrupt rise in interest prompted the development of several Web services in response, offering on-demand geotagged maps [16] of the disaster area to help guide relief efforts. Similarly, efforts were initiated to collect real-time images of the area, which are then composed together piecemeal by services in order to capture more holistic views. But due to their popularity, the availability of such services becomes an issue during this critical time.

However, because service requests during these situations are often related, e.g., displaying a traffic map of a certain populated area in Port-au-Prince, a considerable amount of redundancy among these services can be exploited. Consequently, their derived results can be reused to not only accelerate subsequent queries, but also to help reduce service traffic. Conversely, if the data is cached, but left unused, it would likely incur storage costs that will not be offset by savings on processing costs. As demand for derived data can change over time, it is important to exploit the elasticity of Cloud environments and dynamically provision storage resources.

In this paper, we describe an approach to cache and utilize service-derived results. We implement a cooperative caching framework for storing the services' output data in-memory for facilitating fast accesses. Our system has been designed to automatically scale, and relax, elastic compute resources as needed. We should note that automatic scaling services exist on most Clouds. For instance, Amazon AWS allows users to assign certain rules, e.g., *scale up by one node if the average CPU usage is above 80%*. But while auto-scalers are suitable for Map-Reduce applications [15], among other easily parallelizable applications, in cases where much more distributed coordination is required, elasticity does not directly translate to scalability. Such is the case for our proposed cache, and we have designed and evaluated specific scaling logic for our system. In the direction of the cost-incentivized down-scaling, a *decay*-based cache eviction scheme is implemented for node deallocation. Depending upon the nature of data and services, security and authentication can be important concerns in a system of this nature [22]. Our work targets scenarios where all data and services are shared among users of that particular Cloud environment, and these issues are thus not considered here.

Using a real service to represent our workload, we have evaluated many aspects of the cache extensively over the Amazon

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

EC2 public Cloud. In terms of utilization, the effects of the cache over our dynamic compute node allocation framework has been compared with static, fixed-node models. We also evaluate our system's resource allocation behavior. Overall, we are able to show that our cache is capable obtaining minimal miss rates while utilizing far less nodes than statically allocated systems of fixed sizes in the span of the experiment. Finally, we run well-designed experiments to show our cache's capacity for full elasticity — its ability to scale up, and down, amidst varying workloads over time.

The high-level contributions of this work are as follows. Our cache was originally proposed to speed up computations in our scientific workflow system, *Auspice* [10], [11]. Thus, the cache's API has been designed to allow for transparent integration with Auspice, and other such systems, to compose derived results directly into workflow plans. Our system is thus easily adaptable to many types of applications that can benefit from data reuse. We are furthermore considering cooperative caching in the context of Clouds, where resource allocation and deallocation should be coordinated to harness elasticity. To this end, we implement a sliding window view to capture user interest over time.

The remainder of this paper is organized as follows. In the Section II, we present the background and goals for our cache framework. In Section III, we formalize the structures and methods involved for our system's operation. Experimental results are discussed in Section IV. We identify some related works in Section V and finally conclude in Section VI.

II. System Goals and Design

In this section, we identify several goals and requirements for our system, and we also discuss some design decisions to implement our data cache.

Provisioning Fast Access Methods:

The ability to store large quantities of precomputed data is hardly useful without efficient access. This includes not only identifying which cooperating cache node contains the data, but also facilitating fast hits and misses within that node. The former goal could be achieved through such methods as hashing or directory services, and the latter requires some considerations toward indexing. Although the index structure is application dependent, we utilize well-supported spatial indices [26], [24] due to the wide range of applications that they can accommodate and also their de facto acceptance into most practical database systems. This implies an ease of portability, which relates to the next goal.

Transparency and High-Level Integration with Existing Systems:

Our cache must subscribe to an intuitive programming interface that allows for nonintrusive integration into existing systems. Like most caches, ours should only present high-level *search* and *update* methods while hiding internal nuances from the programmer. These details might include victimization schemes, replacement policies, management of underlying compute resources, data movement, etc. In other words, our system can be viewed as a Cloud service, from the application developer's perspective, for indexing, caching, and reusing precomputed results.

Graceful Adaptation to Varying Workloads:

An increase in service request frequency implies a growing amount of data that must be cached. Taking into consideration the dimensionality of certain data sets, it is easy to predict that caches can quickly grow to sizes beyond main memory as query intensive situations arise. In-core containment of the index, however, is imperative for facilitating fast response times in cache systems. The elastic resource allocation afforded by the Cloud is important here; in these cases, our system should also increase its available main memory to guarantee in-core access. Similarly, a decrease in request frequency should invoke a contraction of currently allocated resources.

A. Design Decisions

First, our cache has been designed under a cooperative scheme, where cache nodes are distributed over the Cloud, and each node stores only a portion of the entire cache. Upon a cache overflow, our system splits the overflown node and migrates its data either to a new allocated Cloud node, or an existing cooperating node. Similarly, our cache should understand when to relax and merge compute nodes to save costs. This approach is somewhat akin to distributed hashtables (DHT) and web proxies, but we state the distinctions in Section V.

Each node in our system employs a variant of B+-Trees [6] to index cached data due to its familiar and pervasive nature. Because B+-Trees are widely accepted in today's database systems, its integration is simplified. Due to this fact, many approaches have been proposed in the past to extend B+-Trees to various application domains, which makes it extremely portable. Because our specific application involves spatiotemporal data sets, we utilize B^x-Trees [26] to index cached data. These structures modify B+-Trees to store spatiotemporal data through a linearization of time and location using space-filling curves, and thus, individual one-dimensional keys of the B+-Tree can represent spatiotemporality.

Another design decision addresses the need to handle changes in the cache's underlying compute structure. The B+-Tree index previously discussed is installed on each cache server in the cooperating system. However, as we explained earlier, due to memory overflow/underflow, the system may have to dynamically expand/contract. Adding and removing cache nodes should take minimal effort, which is a deceptively hard problem. To illustrate, consider an n node cooperative cache system, and each node is assigned a distinct *id*: $0, \ldots, n-1$. Identifying the node responsible for caching some data identified by key, k, is trivial with static hashing, i.e., $h(k) = (k \mod n)$ can be computed as node *id*. Now assume that a new node is allocated, which effectively modifies the hash function to $h(k) = (k \mod n+1)$. This ostensibly simple change forces most currently keyed records to be rehashed and, worse, relocated using the new hash. Rehashing and migrating large volumes of records after each node acquisition is, without saying, prohibitive.

To handle this problem, also referred to as *hash disruption* [36], we implement consistent hashing [29]. In this hashing method, we first assume an auxiliary hash function, e.g., $h'(k) = (k \mod r)$, for some fixed r. Within this range exists a sequence of p buckets, $B = (b_1, \ldots, b_p)$, with each bucket mapped to a single cache

node. Figure 1 (top) represents a framework consisting two nodes and five buckets. When a new key, k, arrives, it is first hashed via the auxiliary hash $h'(k)$ and then assigned to the node referenced by $h(k)$'s closest upper bucket. In our figure, the incoming k is assigned to node n_2 via b_4. Often, the hash line is implemented in a circular fashion, i.e., a key $k \mid b_5 < h'(k) \leq r-1$ would be mapped to n_1 via b_1.

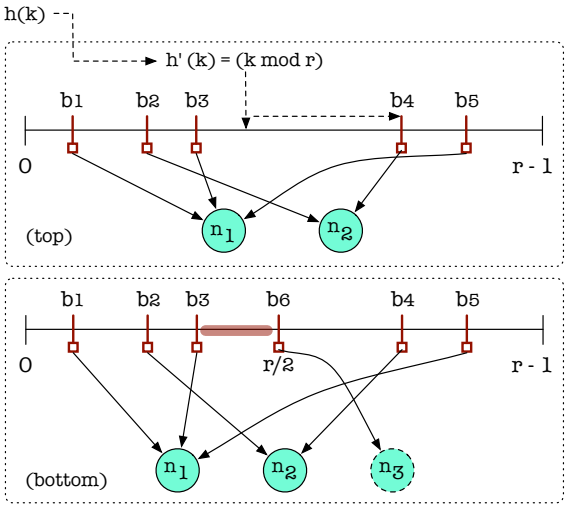


Fig. 1. Consistent Hashing Example

Because the hash map is fixed, consistent hashing reduces hash disruption by a considerable factor. For instance, let us consider Figure 1 (bottom), where a new node, n_3, has been acquired and assigned by some bucket $b_6 = r/2$ to help share the load between b_3 and b_4. The introduction of n_3 would only cause a small subset of keys to be migrated, i.e., $k \mid b_3 < h'(k) \leq b_6$ (area within the shaded region) from n_2 to n_3 in lieu of a rehash of all records. Thus, we can implement the task of supporting elastic Cloud structures without hash disruption.

III. Cache Design and Access

Before presenting cache access methods, we first state the following definitions. Let $N = \{n_1, \ldots, n_m\}$ denote the currently allocated cache nodes. We define $||n||$ and $\lceil n \rceil$ to be the current space used and capacity respectively on cache node n. We further define the ordered sequence of allocated buckets as $B = (b_1, \ldots, b_p)$ such that $b_i \in [0, r)$ and $b_i < b_{i+1}$. Given an auxiliary, fixed hash function, $h'(k) = (k \mod r)$, in a circular implementation, our hash function is defined,

$$h(k) = \begin{cases} b_1, & \text{if } h'(k) > b_p \\ \arg\min_{b_i \in B} b_i - h'(k) : b_i \geq h'(k), & \text{otherwise} \end{cases}$$

For reading comprehension, we have provided a summary of identifiers in Table I. We can now focus on our proposed algorithms for cache access, migration, and contraction over the Cloud. Note that we will not discuss the cache search method, as it is trivial, i.e., by running a B+-Tree search for k on the node referenced by $h(k)$.

TABLE I
Listing of Identifiers

Identifier	Description
k	A queried key
$B = (b_1, \ldots, b_p)$	The list of all buckets on the hash line
N	The set of all nodes in the cooperative cache
$n \in N$	A cache node
$\|\|n\|\|$	Current size of index on node n
$\lceil n \rceil$	Overall capacity on node n
$T = (t_1, \ldots, t_m)$	Sliding window of size m
$t_i \in T$	A single time slice in the sliding window, which records all keys that were queried in that period of time
α	The decay, $0 < \alpha < 1$, used in the calculation of $\lambda(k)$
$\lambda(k)$	Key k's likelihood of being evicted
T_λ	Eviction threshold, i.e., $k \mid \lambda(k) < T_\lambda$ are designated for eviction

A. Insertion and Migration

The procedure for inserting into the cache could invoke migration, which complicates the otherwise simple insertion scheme. In Algorithm 1, the insert algorithm is defined with a pair of inputs, k and v, denoting the key and value object respectively. The Greedy Bucket Allocation (GBA) Insert algorithm is so named as to reflect that, upon node overflows, we greedily consider preexisting cache nodes as the data migration destination. In other words, node allocation is a last-resort option to save cost.

Algorithm 1 GBA-insert(k, v)

```
1:  static NodeMap[...]
2:  static B = (...)
3:  static h' : K → [0, r)
4:  n ← NodeMap[h'(k)]
5:  if ||n|| + sizeof(v) < ⌈n⌉ then
6:      n.insert(k, v)        ▷ insert directly on node n
7:  else
8:      ▷ n overflows
9:      ▷ find fullest bucket referencing n
10:     b_max ← argmax_{b_i ∈ B} ||b_i|| ∧ NodeMap[b_i] = n
11:     k^μ ← μ(b_max)
12:     n_dest ← n.sweep-migrate(min(b_max), k^μ)
13:     ▷ update structures
14:     B ← (b_1, ..., b_i, h'(k^μ), b_{i+1}, ..., b_p) | b_i < h'(k^μ) < b_{i+1}
15:     NodeMap[h'(k^μ))] ← n_dest
16:     GBA-insert(k, v)
17: end if
```

On Line 1, the statically declared inverse hash map is brought into scope. This structure defines the relation $NodeMap[b] = n$ where n is the node mapped to bucket value b. The ordered list of buckets, B, as well as the auxiliary consistent hash function, h', are also brought into scope (Lines 2-3). After identifying k's bucket and node (Line 4), the (k, v) pair is inserted into node n if the system determines that its insertion would not cause a memory overflow on n (Lines 5-6). Since cache indices expanding into disk memory would become prohibitively slow, when an overflow is detected, migration of portions of the index must be invoked to make space (Line 7).

The goal of migration is to introduce a new bucket into the overflown interval that would reduce the load of about half of

the keys from the overflown bucket. However, the fullest bucket may not necessarily be b. On (Line 10), we identify the fullest bucket which references n, then invoke the migration algorithm on a range of keys, to be described shortly (Line 11-12). As a simple heuristic, we opt to move approximately half the keys from bucket b_{max}, starting from the lowest key to the median, k^{μ}. The sweep-and-migrate algorithm returns a reference to the node (either preexisting or newly allocated), n_{dest}, to which the data from n has been migrated. On (Lines 13-15), the buckets, B, and node mapping data structures, $NodeMap[\ldots]$, are updated to reflect internal structural changes. Specifically, a new bucket is created at $h'(k^{\mu})$ and it references n_{dest}. The algorithm is finally invoked recursively to attempt proper insertion under the modified cache structure.

The *Sweep-and-Migrate* function, shown in Algorithm 2, resides on each individual cache server, along with the indexing logic. As an aside, in our implementation, the cache server is automatically fetched from a remote location on the startup of a new Cloud instance. The algorithm inputs the range of keys to be migrated, k_{start} and k_{end}. The least loaded node is first identified from the current cache configuration (Line 1). If it is projected that the key range cannot fit within n_{dest}, then a new node must be allocated from the Cloud (Lines 2-5). The aggregation test (Line 2) can be done by maintaining an internal structure on the server which holds the keys' respective object size.

Once the destination node has been identified we begin the transfer of the key range. We now describe the approach to find and *sweep* all keys in the specified range from the internal B+-Tree index. The B+-Tree's linked leaf structure simplifies the record sweep portion of our algorithm. First, a search for k_{start} is invoked to locate its leaf node (Line 9). Then, recalling that leaf nodes are arranged as a key-sorted linked list in B+-Trees, a sweep (Line 10-22) on the leaf level is performed until k_{end} has been reached. For each leaf visited, we transfer all associated (k, v) record to n_{dest}.

Algorithm 2 sweep-migrate(k_{start}, k_{end})

```
1: n_dest ← argmin_{n_i ∈ N} ||n_i||
2: ▷ stolen keys and values will overflow n_dest
3: if ||n_dest|| + Σ_{k=k_start}^{k_end} sizeof(k,v) > ⌈n_dest⌉ then
4:     n_dest ← nodeAlloc()
5: end if
6: ▷ manipulate B+-Tree index and transfer to n_dest
7: end ← false
8: ▷ L = leaf initially containing k_start
9: L ← btree.search(k_start)
10: while (¬end ∧ L ≠ NULL) do
11:     ▷ each leaf node contains multiple keys
12:     for all (k, v) ∈ L do
13:         if k ≤ k_end then
14:             n_dest.insert(k, v)
15:             btree.delete(k)
16:         else
17:             end ← true
18:             break
19:         end if
20:     end for
21:     L ← L.next()
22: end while
23: return n_dest
```

Analysis of GBA-Insert

GBA-insert is difficult to generalize due to variabilities of the system state, which can drastically affect the runtime behavior of migration, e.g., number of buckets, migrated keys, size of each object, etc. To be succinct in our analysis, we make the simple assumption that $sizeof((k, v)) = 1$ to normalize cached records. This simplification also allows us to imply an even distribution over all buckets in B and nodes in N. In the following, we only consider the worst case.

We begin with the analysis of sweep-and-migrate (Algorithm 2), whose time complexity is denoted $T_{migrate}$. First, the maximum number of keys that can be stolen from any node is half of the record capacity of any node: $\lceil n \rceil / 2$. This is again due to our assumption of an even bucket/node distribution, which would cause Algorithm 1's calculation of $min(b_{max})$ and k^{μ} to be assigned such that $min(b_{max}) - k^{\mu} \approx \lceil n \rceil / 2$, and thus the *sweep* phase can be analyzed as having an $O(\log_2 ||n||)$-time B+-Tree search followed by a linear sweep of $\lceil n \rceil / 2$ records, i.e.,

$$\log_2 ||n|| + \lceil n \rceil / 2$$

The complexity of $T_{migrate}$, then, is the sum of the above sweep time and the time taken to move the worst case number of records to another node. If we let T_{net} denote the time taken to move one record,

$$T_{migrate} = \log_2 ||n|| + \lceil n \rceil / 2 (T_{net} + 1)$$

We are now ready to solve for T_{GBA}, the runtime of Algorithm 1. As noted previously, $h(k)$ can be implemented using binary search on B – the ordered sequence of p buckets, i.e., $T(h(k)) = O(\log_2 p)$. After the initial hash function is invoked, the algorithm enters the following cases: (i) the record is inserted trivially, or (ii) a call to migrate is made before trivially inserting the record (which requires a subsequent hash call). That is,

$$T_{GBA} = \begin{cases} \log_2 p, & \text{if } ||n|| + 1 < \lceil n \rceil \\ 2 \log_2 p + T_{migrate}, & \text{otherwise} \end{cases}$$

Finally, after substitution and worst case binding, we arrive at the following conditional complexity due to the expected dominance of record transfer time, T_{net},

$$T_{GBA} = \begin{cases} O(1), & \text{if } ||n|| + 1 < \lceil n \rceil \\ O((\lceil n \rceil / 2) T_{net}), & \text{otherwise} \end{cases}$$

Although T_{net} is neither uniform nor trivial in practice, our analysis is sound as actual record sizes would likely increase T_{net}. But despite the variations on T_{net}, the bound for the latter case of T_{GBA} remains consistent due to the significant contribution of data transfer times.

B. Cache Eviction

Consider the situation when some interesting event/phenomenon causes a barrage of queries in a very short amount of time. Up till now, we have discussed methods for scaling our cache system *up* to meet the demands of these query-intensive circumstances. However, this demanding period may abate over time, and the resources provisioned by our system often become superfluous. In traditional distributed (e.g.,

cluster and grid) environments, this was less of an issue. For instance, in advance reservation schemes, resources are reserved for some fixed amount of time, and there is little incentive to scale back down. In contrast, the Cloud's usage costs prompts an important motivation to scale our system down.

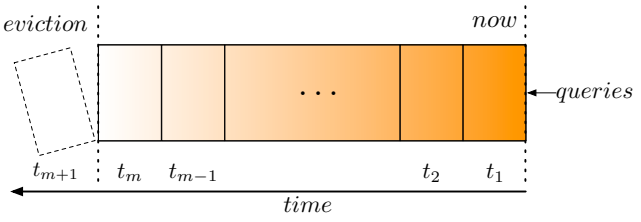


Fig. 2. Sliding Window of the Most Recently Queried Keys

We implement a cache contraction scheme to merge nodes when query intensities are lowered. Our scheme is based on a combination of exponential decay and a temporal sliding window. Because the size of our cache system (number of nodes) is highly dependent on the frequency of queries during some timespan, we propose a global cache eviction scheme that captures querying behavior. In our contraction scheme, we employ a streaming model, where incoming query requests represent streaming data, and a global view of the most recently queried keys is maintained in a sliding window. Shown in Figure 2, our sliding window, $T = (t_1, \ldots, t_m)$, comprises m time slices of some fixed real-time length. Each time slice, t_i, associates a set of keys queried in the duration of that slice. We argue that, as time passes, older unreferenced keys (i.e., those in the lighter shaded region, t_i nearing t_m) should have a lower probability of existing in the cache. As these less relevant keys become evicted, the system makes room for newer, incoming keys (i.e., those in the darker shaded region, t_i nearing t_1) and thus capturing temporal locality of the queries.

Cache eviction occurs when a time slice has reached t_{m+1}, and at this time, an eviction score,

$$\lambda(k) = \sum_{i=1}^{m} \alpha^{i-1} |\{k \in t_i\}|$$

is computed for every key, k, within the expired slice. The ratio, $\alpha : 0 < \alpha < 1$, is a *decay* factor, and $|\{k \in t_i\}|$ returns the number of times k appears in some slice t_i. Here, α is passive in the sense that a higher value corresponds to a larger amount of keys that is kept in the system. After λ has been computed for each key in t_{m+1}, any key whose λ falls below the threshold, T_λ, is evicted from the system. Notice that α is amortized in the older time slices, in other words, recent queries for k are rewarded, so k is less likely to be evicted. Clearly, the sliding window eviction method is sensitive to the values of α and m. A baseline value for T_λ would be α^{m-1}, which will not allow the system to evict any key if it was queried even just once in the span of the sliding window. We will show their effects in the experimental section.

Due to the eviction strategy, a set of cache nodes may eventually become lightly loaded, which is an opportunity to scale our system down. The nodes' indices can be merged, and subsequently, the superfluous node instances can be discarded. When a time slice expires, our system invokes a simple heuristic for *contraction*. Our system monitors the memory capacity on each node. After each interval of ϵ slice expirations, we identify the two least loaded nodes and check whether merging their data would cause an overflow. If not, then their data is migrated using methods tantamount to Algorithm 2.

Analysis of Eviction and Contraction

The contraction time is the sum of eviction time and node merge time, $T_{contract} = T_{evict} + T_{merge}$. To analyze merge time, we first note that it takes $O(1)$ time to identify the two least loaded nodes, as we can dynamically maintain a list of nodes sorted by capacity. If the data merge is cannot be performed, the algorithm simply halts. On the other hand, it executes a slight variant of the *Sweep-and-Migrate* algorithm to move the index from one node to another, which, combined with our previous analysis of $T_{migrate}$, is $||n_{min}||(T_{net} + 1)$ where $||n_{min}||$ is the size of the migrated index. If we ignore the best case $O(1)$ time expended when contraction is infeasible, then the time taken by T_{merge} can be summarized as follows,

$$T_{merge} = ||n_{min}||(T_{net} + 1)$$

The contraction method is invoked every ϵ time slices' expiration from the sliding window. By itself, the sliding window's slice eviction method, T_{evict} can be summarized by $T_{evict} = mK$ where m is the size of the sliding window, and $K = |\{k \in t_{m+1}\}|$ is the total number of keys in the evicted time slice, t_{m+1}. However, since T_{evict} again pales against network traffic time, T_{net}, its contribution can be assumed trivial. Together, the overall eviction and contraction method can be bound $T_{contract} = O(||n_{min}||T_{net})$.

IV. Experimental Results

In this section, we discuss the evaluation of our derived data cache system. We employ the Amazon Elastic Compute Cloud (EC2) to support all of our experiments.

A. Experimental Setup

Each Cloud node instance runs an Ubuntu Linux image on which our cache server logic is installed. Each image runs on a *Small EC2 Instance*, which, according to Amazon, comprises 1.7 GB of memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor) on a 32-bit platform. In all of our experiments, the caches are initially cold, and both index and data are stored in memory.

As a representative workload, we executed repeated runs of a *Shoreline Extraction* query. This is a real application, provided to us by our colleagues in the Department of Civil and Environmental Engineering and Geodetic Science here at Ohio State University. Given pair of inputs: location, L, and time of interest, T, this service first retrieves a local copy of the Coastal Terrain Model (CTM) file with respect to (L, T). To enable this search, each file has been indexed via their spatiotemporal metadata. CTMs contain a large matrix of a coastal area where each point denotes a depth/elevation reading. Next, the service retrieves actual water level readings, and finally given the CTM and water level, the coast line is interpolated and returned. The baseline execution time of this service, i.e., when executed without any

caching, typically takes approximately 23 seconds to complete, and the derived shoreline result is $<$ 1kb.

We have randomized inputs over 64K possibilities for each service request, which emulates the worst case for possible reuse. The 64K input keys represent linearized coordinates and date (we used the method described in B^x-Trees [26]). The queries are first sent to a coordinating compute node, and the underlying cooperating cache is then searched on the input key to find a replica of the precomputed results. Upon a hit, the results are transmitted directly back to the caller, whereas a miss would prompt the coordinator to invoke the shoreline extraction service.

In the following experiments, in order to regulate the integrity in querying rates, we submitted queries with the following loop:

```
for time step i ← 1 to ... do
    R ← current query rate(i)
    for j ← 1 to R do
        invoke shoreline service(rand_coordinates())
    end for
end for
```

Specifically, we invoke R queries per *time step*, and thus each time step does not reflect real time. Note that the granularity of a time step in practice, e.g., t seconds, minutes, or hours, does not affect the overall hit/miss rates of the cache. At each time step, we observed and recorded the average service execution time (in number of seconds real time), the number of times a query reuses a cached record (i.e., hits), and the number of cache misses.

B. Evaluating Cache Benefits

The initial experiment evaluates the effects of the cache without node contraction. In other words, the length of our eviction sliding window is ∞. Under this configuration, our cache is able to grow as large as it needs to handle the size of the cache. We run our cache system over static, fixed-node configurations (*static-2, static-4, static-8*), comparable to current cluster/grid environments, where the amounts of nodes one can allocate is typically fixed. The fixed-node settings subscribe to the simple LRU eviction policy. We compare these static versions against our approach, Greedy Bucket Allocation (*GBA*), which runs over the EC2 public Cloud. For these experiments, we submitted one query per time step, i.e., the query submission loop is configured $R = 1$ over 2×10^5 time steps.

We executed the shoreline service repeatedly with varying inputs. Figure 3, which shows the respective relative speedups over the query's actual execution time. We observed and plotted the speedup for every $I = 25000$ queries elapsed in our experiment. Expectedly, the speedup provided by the static versions flatten somewhat quickly, again due to the nodes reaching capacity. The relative speedups converge at $1.15\times$ for *static-2*, $1.34\times$ for *static-4*, and $2\times$ for *static-8*. *GBA*, on the other hand, was capable of achieving a relative speedup of over $15.2\times$. Note that the speedup in Figure 3 is shown in $\log_{10}$-scale.

The node allocation behavior (shown against the right y-axis) shows that *GBA* allocates 15 nodes in the end of the experiment. But since allocation was only invoked as a last resort on-demand option, $\lceil 12.6 \rceil = 13$ nodes were utilized, if averaged over the lifespan of this experiment. This translates to less overall EC2

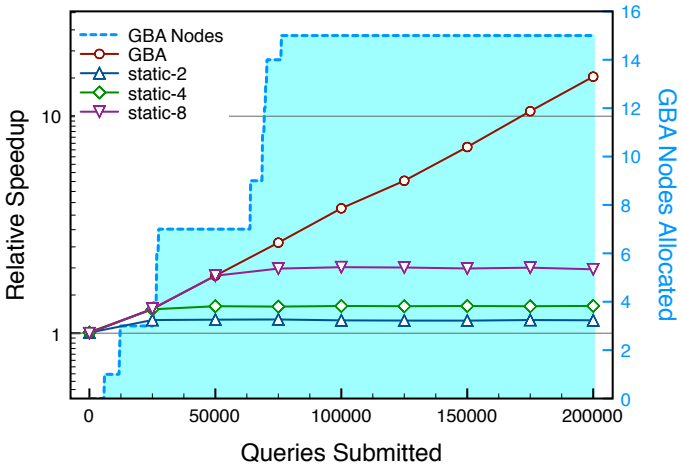


Fig. 3. Speedups Relative to Original Service Execution

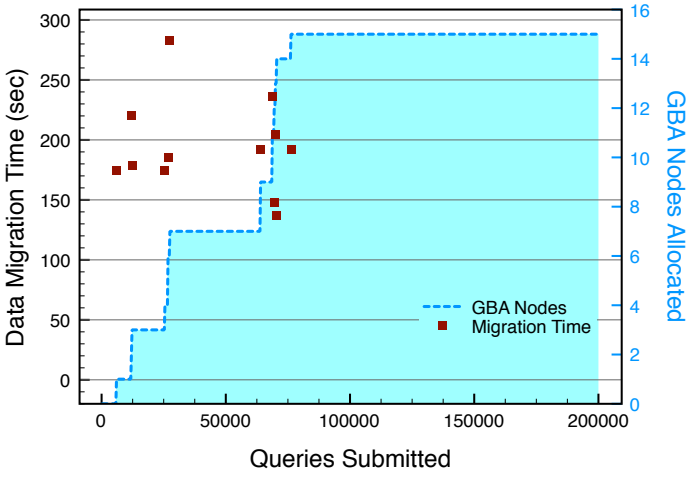


Fig. 4. *GBA* Migration Times

usage cost per performance over static allocations. The growth of nodes is also not unexpected, though, at first glance it appears to be exponential. Early into the experiment, the cooperating cache's overall capacity is initially too small to handle the query rate, until stabilizing after $\sim$ 75000 queries have been processed.

Next, we summarize in Figure 4 the overhead of node splitting (upon cache overflows) as the sum of node allocation and data migration times for *GBA*. It is clear from this figure that this overhead can be quite large. Although not shown directly in the figure, we note it is the node allocation time, and not the data movement time, which is the main contributor to this overhead. However, these penalties are amortized because node allocation is seldom invoked. We also posit that the demand for node allocation diminishes as the experiment proceeds even with high querying rates due to system stabilization. Moreover, techniques, such as asynchronous preloading of EC2 instances and replication, can also be used to further minimize this overhead, although these have not been considered in this paper.

C. Evaluating Cache Eviction and Contraction

Next, we evaluate our eviction and contraction scheme. Two separate experiments were devised to show the effects of the sliding window and to show that our cache is capable of relaxing resources when feasible. We randomize the query inputs points over 32K possibilities, and we generated a workload to simulate a *query intensive situation*, such as the one described in the introduction in the following manner. Recall, from the query submission loop we stated early in this section, that a *time step* denotes an iteration where R queries are submitted. Specifically, in the following experiments, for the first 100 time steps, the querying rate is fixed at $R = 50$ queries/time step. From 101 to 300 time steps, we enter an intensive period of $R = 250$ queries/time step to simulate heightened interest. Finally, from 400 time steps onward, the query rate reduced back down to $R = 50$ queries/time step to simulate waning interest.

We show the relative speedup for varying sliding window sizes of $m = 50$ time steps, $m = 100$ time steps, $m = 200$ time steps, and $m = 400$ time steps in Figures 5(a), 5(b), 5(c), and 5(d) respectively. Recall that the sliding window will attempt to maintain, with high probability, all records that were queried in the m most recent time steps. To ensure this probability, the decay has been fixed at $\alpha = 0.99$ for these experiments, and the eviction threshold is set at the baseline $T_\lambda = \alpha^{m-1} \approx 0.367$ to avoid evicting any key which had been queried even just once within the window.

From these figures, we can observe that our cache elastically adapts to the query-intensive period by improving overall speedup, albeit to varying degrees depending on m. For example, the maximum observable speedup achieved with the smaller sized window in Figure 5(a) is approximately $1.55\times$, with an average node allocation of $\lceil 1.7 \rceil = 2$ nodes. In contrast, the much larger sliding window of 400 in Figure 5(d) offers a maximum observable speedup of $8\times$, with an average use of $\lceil 5.6 \rceil = 6$ nodes. We can also observe that, after the query intensive period expires at 300 time steps, the sliding window will detect the normal querying rates and remove nodes as they become superfluous. This trend can also be seen in all cases — nodes do not decrease back down to 1 because our contraction algorithm is quite conservative. We have set our node-merge threshold to 65% of space required to store the coalesced cache to address *churn-avoidance*, i.e., repeated allocation/deallocation of nodes.

In terms of performance, our system benefits from higher querying rates, as it populates our cache faster within the window. The noticeable performance disparities among the juxtaposed figures also indicate that the size of the sliding window is a highly determinant factor on both performance and node allocation, i.e., cost. Compared with the ∞ sliding window experiments in Figure 3, we can observe that our eviction scheme affords us comparable results with lesser amounts of nodes, which translates to smaller cost of compute resource provisioning in the Cloud.

For these same experiments, we analyze the eviction and data reuse and eviction behavior over time in Figures 6(a), 6(b), 6(c), and 6(d). One can see that, invariably, reuse expectedly increase over the query-intensive period, again to varying degrees depending on window size. After 300 time steps into the experiment, the query rate resumes to $R = 50$/time step, which means less chances for reuse. This allows aggressive eviction behaviors in all cases, except in Figure 6(d), where the window extends beyond 300 time steps.

There are several interesting trends that can be seen in these experiments. First, the eviction behavior in Figure 6(d) appears to oppose the upward trend observed in all other cases. Due to the size of this window, the decay becomes extremely small near the evicted time slice, and our cache removes records quite aggressively. At the same time, this eviction behavior decreases over time due to the evicted slices being a part of the query-intensive period, which accounted for more reuse, and thus, less probability for eviction. This trend simply was not seen in all other cases because the window size did not allow for such probability for reuse before records were candidates for eviction.

Another interesting observation can be made on node growth between Figures 6(c) and 6(d). Notice that node allocation continues to increase well after the intensive period in Figure 6(d) due to its larger window size. While this ensures more hits after the query-intensive period expires, justifying the tradeoff of allocation cost and the speedup of the queries after 300 time steps is questionable in this scenario. This implies that a dynamic window size can be employed here to optimize costs, which we plan to address in future works.

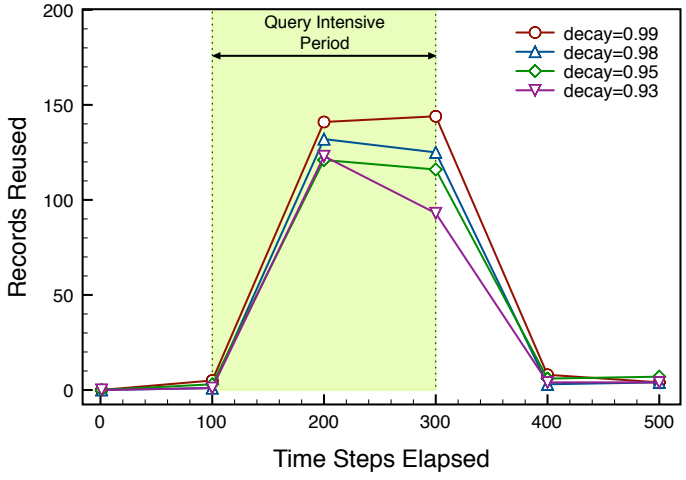
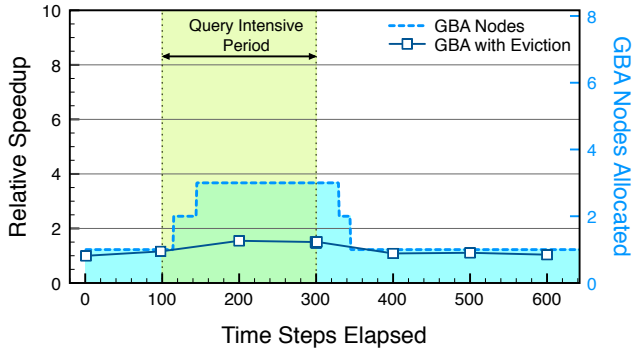


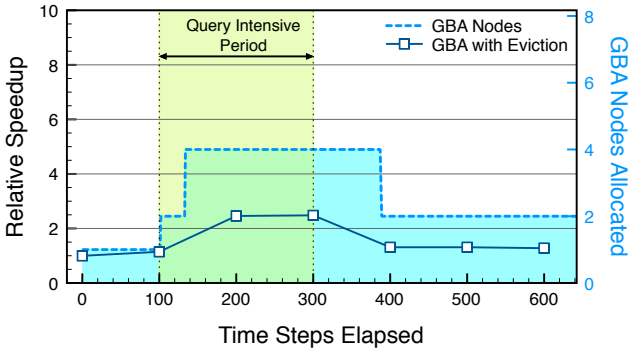
Fig. 7. Data Reuse Behavior for Various Decay $\alpha = 0.99, 0.98, 0.95, 0.93$

Finally, we present the effects of the decay, α, on cache eviction behavior. We used same querying configuration as in the above sliding window experiments, where normal querying rate is $R = 50$ queries/time step, and the intensive rate is $R = 250$ queries/time step. We evaluated the eviction mechanism under the $m = 100$ sliding window configuration on four decay values: $\alpha = 0.99, 0.98, 0.95, 0.93$. We would expect that a smaller decay value would lead to more aggressive eviction, which can be inferred from Figure 7. Also note the sensitivity of α due to its exponential nature.

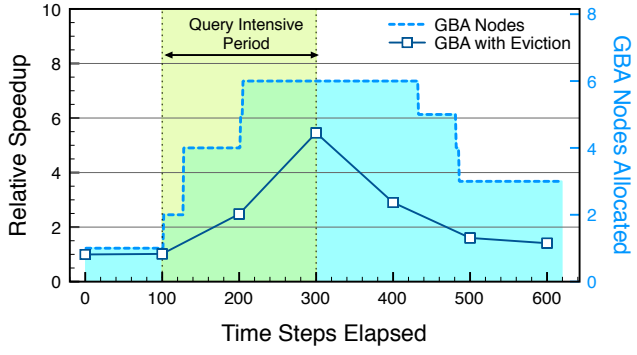
When decay is small, a certain record must be reused many more times to be kept cached in the window. However, the benefit of this can also be argued from the perspective of cost – the cache system pertaining to a smaller α grows much slower and, according to Figure 7, the number of actual cache hits



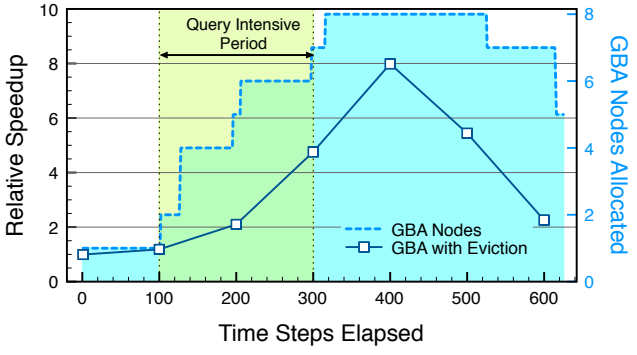
(a) Sliding Window Size = 50 time steps



(b) Sliding Window Size = 100 time steps

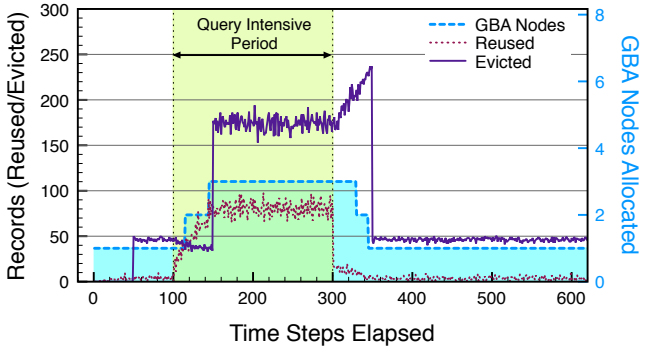


(c) Sliding Window Size = 200 time steps

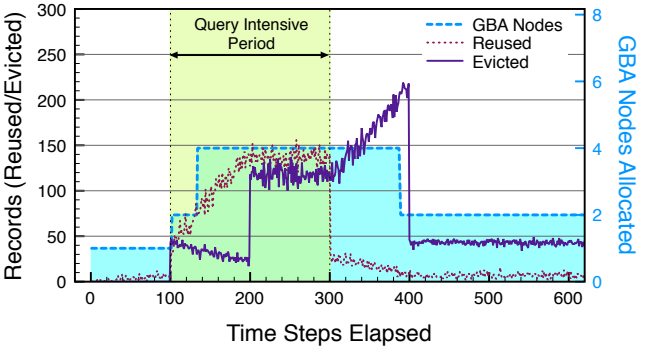


(d) Sliding Window Size = 400 time steps

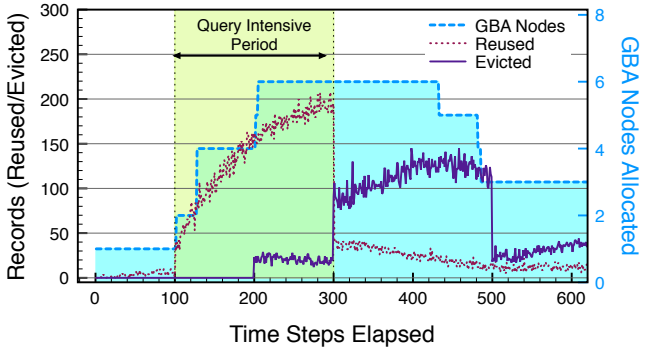
Fig. 5. Speedup under Eviction/Contraction (Normal Query Rate = 50 queries/time step, Intensive Rate = 250 queries/time step)



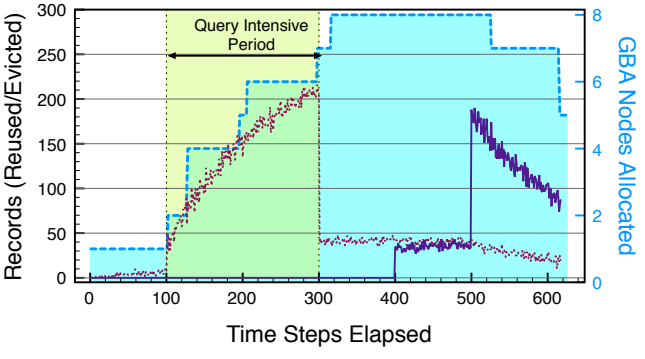
(a) Sliding Window Size = 50 time steps



(b) Sliding Window Size = 100 time steps



(c) Sliding Window Size = 200 time steps



(d) Sliding Window Size = 400 time steps

Fig. 6. Data Reuse and Eviction Behavior (Normal Query Rate = 50 queries/time step, Intensive Rate = 250 queries/time step)

over this execution does not seem to vary enough to make any extraordinary contribution to speedup.

D. Summary and Discussion

We have evaluated our cooperative cache system from various perspectives. The relative performance gains from the infinite eviction window experiments show that caching service results over the Cloud is a fruitful endeavour, but it comes at the expense of high node allocation for ensuring cache capacity. We showed that the overhead of node splitting can be quite high, but is so seldom invoked that its penalties are amortized over the sheer volume of queries submitted. We also argue that it is rarely invoked once the cache's capacity stabilizes. However, this prompts a need for more intelligent strategies for reducing node allocation penalties.

Our sliding window-based eviction strategy appears to offer a good compromise between performance and cost tradeoffs, and captures situations with heightened (and waning) query intensities. For instance, the larger $m = 400$ sliding window, shown in Figure 6(d), achieves an $8\times$ speedup at the peak of the query intensive period, while only requiring a maximum of 8 nodes, which further reduces down to 5 nodes toward the end of the experiment.

Finally, through a study of eviction decay, we are able to conclude that both system parameters, α and sliding window size m, account for node growth (and thus, cost) and performance. However, it is m that contributes far more significantly to our system. A dynamically changing m can thus be very useful in driving down cost.

We have also assessed the various cost aspects of the Cloud's persistent storage, such as Amazon S3 and Elastic Block Storage (EBS), and other machine instance-types in our cache framework. The cost varies among the added benefits of data persistence and machine instances with higher bandwidth and memory. But because this paper deals mostly with the performance aspect, and with respect to space constraints, we discuss our findings of cost benefits and performance tradeoffs among the varying Amazon Cloud storage types in a related paper [9].

V. Related Works

Research efforts in storage management have proposed a cache layer for alleviating long access times to persistent storage. For instance, Cardenas et al.'s uniform, collaborative cache service [8] and Tierney et al.'s Distributed-Parallel Storage System (DPSS) [48] offer a buffer between clients and access to mass storage systems including SDSC's Storage Resource Broker [5]. Other efforts, including works done by Otoo et al. [35], Bethel et al. [7], and Vazhkudai et al. [49], consider these intermediate caching issues in various storage environments for scientific computing. Work has also been produced in the direction of optimal replacement policies for disk caching in data grids [27]. Other utilities have sought for the storage of more detailed information on the scientific, such as virtual data traces, known as provenance. Chimera [19] is a system that stores information on virtual data sets which affords scientists a way to understand how certain results can be derived, as well as a way to reproduce data derivations.

Tangential to our work is *multi-query optimization* (MQO) in the area of databases, where related queries may share, and can benefit from reusing, common data. Methods for processing like-queries together, rather than independently, could thus greatly improve performance [41]. Toward this goal, *semantic caching* [39], i.e., based on semantic locality, has been considered in past efforts. Particularly, Andrade, *et al.* describe an active semantic caching middleware to frame MQO applications in a grid environment [2]. This middleware combines a proxy service with application servers for processing, which dynamically interacts with cache servers. The proxy acts as a coordinator, composing suitable schedules for processing the query over the supported environment. Our system differs, like all the aforementioned effort, in that it considers cost-based pressures of the Cloud. Specifically, our cache is sensitive to *user interest* over time, and it allocates compute resources to improve performance during query intensive periods, only to relax the resources later to save costs.

The recently proposed *Circulate* architecture employs Cloud proxies for speeding up workfow applications [51], [4]. In their work, proxies close to the computation are used to store intermediate data. This data is then directly routed to the nodes involved in the next stage of the computation. While their overarching goal of reducing composite service time is tantamount to ours, we clarify the distinctions. Their system focuses on eluding unnecessary data transfers to and from some orchestrating node. Ours deliberately caches service results to accelerate processing times under a *query-intensive* scenario. Our work also focuses on strategies for caching, managing, and altering underlying Cloud structures to optimize the availability of cached results under these circumstances.

Memcached [33] is a distributed cache system designed to accelerate general Web applications, storing data-structure agnostic objects in memory of up to 1MB in size. The memcached servers utilize an LRU eviction policy upon reaching capacity. In contrast, our system is capable of expanding to avoid capacity misses during query intensive periods, and relaxes when the period diminishes. We believe memcached can be used in conjunction to our system to exploit the Cloud, and it is worth exploring in the future.

Resource allocation is another related issue. Traditionally, a user requests a provision for some fixed set number of compute resources and reserve a span of time for exclusive usage. A common way to request for resources is batch scheduling, which is a ubiquitous mechanism for job submissions at most supercomputing or Grid sites [44]. Condor [32], [20] manages a distributed compute pool of otherwise idle machines, and puts them to use. Condor allocates resources for jobs based on a "matchmaking" approach [38]. Machines in the pool advertise their resource specifications as well as conditions under which it would be willing to take on a job. A submitted job must also advertise its needs, and Condor allocates the necessary resources by matching machines based on these specifications.

In [28], Juve and Deelman discussed the consequences of applying current resource provisioning approaches on the Grid/Cloud and argued that traditional queue-based reservation models can suffer massive delays due to the heterogeneity of Grid sites (different rules, priority, etc.). Raicu, *et al.*'s Falkon

framework [37] describes an ad hoc resource pools which are preemptively allocated from disparate Grid sites by submitting a provisioning job. User applications submit jobs directly to the provisioner, rather than to the external site with the usual methods. Since the provisioner has already preemptively allocated the necessary resources, overheads of job submissions and dispatch are avoided. These advance reservation schemes, however, abstracts the actual provisioning of resources. As per Sotomayor, *et al.*'s observation, "...resource provisioning typically happen[s] as a side-effect of job submission" [45]. In their paper, they describe a lease-based approach toward resource provisioning, in an effort which seemingly precurses today's Cloud-usage methods, by leveraging virtual machine management. Singh, *et al.*'s provisioning model selects a set of resources to be provisioned that optimizes the application while minimizing the resource costs [43]. Providers advertise slots to the users, and each slot denotes the availability of resources, (e.g., number of processors), which can be reserved for a certain timeframe, for a price. This allows application schedulers to optimize resource provisioning for their application based on cost. In a related effort, Huang, *et al.*'s scheme helps users by automatically selecting which resources to provision on a given workflow (DAG-based) application [25]. Our system exploits on-demand elastic resource management provided by the Amazon EC2 Cloud. In this paper, we have proposed algorithms to scale and relax compute resources to handle varying workloads driven by user interest.

Our system is much-inspired by efforts done in the general area of Web page caching and distributed hashtables (DHT). Several methods can be used to evenly distribute the load among cooperating Web caches, also known as proxies. Gadde, Chase, and Rabinovich's CRISP proxy [21] utilizes a centralized directory service to track the exact locations of cached data. But this simplicity comes at the cost of scalability, i.e., adding new nodes to the system causes all data to be rehashed. Efforts, such as Karger, *et al.*'s consistent hashing [29], [30] have been used to reduce this problem down to only rehashing a subset of the entire data set. Also a form of consistent hashing, Thaler and Ravishankar's approach maps an object name consistently to the same machine [47]. Karger, *et al.*'s technique is currently employed in our cooperative cache. Similar methods have also been used in DHTs to manage large numbers of nodes (e.g., *peer-to-peer* systems) that share data. Chord [46], Pastry [40], and Tapestry [52] are representative of such systems, among others, and can offer log-based guarantees on the number of hops necessary for locating data in such a large-scale system. Because of the volatility, e.g., *churn*, in P2P systems, most DHT-based implementations do not focus on offering transient data availability when a node disconnects, which is crucial to our application scenario, albeit that we concede to the overhead of migration costs.

Our system differs from traditional Web caching and DHT systems in the following ways. The cache was originally developed as a component to our service-based workflow composition system, *Auspice* [10], [11]. We are reusing derived results to accelerate long running services with the side-effect of reducing service load. Additionally, our cache's API allows Auspice to easily compose derived results into complex workflow structures, which requires more appropriate indexing of intermediate data. We are furthermore considering the cooperative cache problem in the context of Clouds, where resource allocation and deallocation must be coordinated to reduce cost.

VI. Concluding Remarks

Cloud providers have begun offering users at-cost access to on demand computing infrastructures. In this paper, we propose a Cloud-based cooperative cache system for reducing execution times of data-intensive processes. The resource allocation algorithm presented herein are cost-conscious as not to over-provision Cloud resources. We have evaluated our system extensively, showing that, among other things, our system is scalable to varying high workloads, cheaper than utilizing fixed networking structures on the Cloud, and effective for reducing service execution times.

A costly overhead is the node allocation process itself. Strategies, such as preloading and data replication can certainly be used to implement an *asynchronous* node allocation. Works on instantaneous virtual machine boots [13], [31] have also been proposed and can be considered here. However, with the current reliance on commercial-grade Clouds, we should seek unintrusive schemes. Modifications to current approaches, like the Falkon Framework [37], where ad hoc resource pools are preemptively allocated from remote sites, may be also employed here. Record prefetching from a node that is predictably close to invoking migration can also be considered to reduce migration cost.

As discussed in Section IV, although our sliding window size for eviction is a parameter to the system, there may be merit in managing this value dynamically to reduce unnecessary (or less cost-effective) node allocation. Predictive eviction methods could be well worth considering.

As we move forward in the data-intensive age, the Cloud has made a timely entrance as a paradigm for supporting such large-scale computing tasks. In this paper, we have explored one such method, through on-demand cooperative caching, for reducing processing times.

Acknowledgments

We would like to thank our reviewers, whose insightful comments and suggestions for are not only helpful, but also crucial, to further development of our research.

This work is supported by NSF grants 0541058, 0619041, and 0833101.

References

[1] Amazon elastic compute cloud, http://aws.amazon.com/ec2.

[2] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments. *Parallel Comput.*, 33(7-8):497–520, 2007.

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

[4] A. Barker, J. B. Weissman, and J. I. van Hemert. The circulate architecture: Avoiding workflow bottlenecks caused by centralised orchestration. *Cluster Computing*, 12(2):221–235, 2009.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Software pioneers: contributions to software engineering*, pages 245–262, 2002.

[7] W. Bethel, B. Tierney, J. lee, D. Gunter, and S. Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, USA, 2000.

[8] Y. Cardenas, J.-M. Pierson, and L. Brunie. Uniform distributed cache service for grid computing. *International Workshop on Database and Expert Systems Applications*, 0:351–355, 2005.

[9] D. Chiu and G. Agrawal. Evaluating caching and storage options on the amazon web services cloud. In *Proceedings of the 11th ACM/IEEE International Conference on Grid Computing (Grid'10)*, 2010.

[10] D. Chiu, S. Deshpande, G. Agrawal, and R. Li. Composing geoinformatics workflows with user preferences. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'08)*, New York, NY, USA, 2008.

[11] D. Chiu, S. Deshpande, G. Agrawal, and R. Li. Cost and accuracy sensitive dynamic workflow composition over grid environments. In *Proceedings of the 9th ACM International Conference on Grid Computing (Grid'08)*, 2008.

[12] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[14] C.Hoffa, G.Mehta, T.Freeman, E.Deelman, K.Keahey, B.Berriman, and J.Good. On the use of cloud computing for scientific workflows. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 640–645. IEEE Computer Society, 2008.

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

[16] Erdas client, http://apollopro.erdas.com/apollo-client.

[17] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[18] I. Foster. Service-oriented science. In *Science, 308*, pages 814–817, 2005.

[19] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.

[20] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[21] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. *Workshop on Hot Topics in Operating Systems*, 0:93, 1997.

[22] R. Geambasu, S. D. Gribble, and H. M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[23] Google app engine, http://code.google.com/appengine.

[24] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[25] R. Huang, H. Casanova, and A. A. Chien. Automatic resource specification generation for resource selection. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[26] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+tree-based indexing of moving objects. In *Proceedings of Very Large Databases (VLDB)*, pages 768–779, 2004.

[27] S. Jiang and X. Zhang. Efficient distributed disk caching in data grid management. *IEEE International Conference on Cluster Computing (CLUSTER)*, 2003.

[28] G. Juve and E. Deelman. Resource provisioning options for large-scale scientific workflows. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 608–613, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[31] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *3rd European Conference on Computer Systems (Eurosys)*, Nuremberg, Germany, April 2009.

[32] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[33] memcached, a distributed memory object caching system, http://www.memcached.org.

[34] Microsoft azure, http://www.microsoft.com/windowsazure/.

[35] E. J. Otoo, D. Rotem, A. Romosan, and S. Seshadri. File caching in data intensive scientific applications on data-grids. In *First VLDB Workshop on Data Management in Grids*. Springer, 2005.

[36] M. Rabinovich and O. Spatschek. *Web caching and replication*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[37] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[38] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, 1999.

[39] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):192–210, 2003.

[40] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[41] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[42] Y. Simmhan, R. Barga, C. van Ingen, E. Lazowska, and A. Szalay. Building the trident scientific workflow workbench for data management in the cloud. *Advanced Engineering Computing and Applications in Sciences, International Conference on*, 0:41–50, 2009.

[43] G. Singh, C. Kesselman, and E. Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 117–126, New York, NY, USA, 2007. ACM.

[44] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *In Proceedings of IPDPS 2000*, pages 127–132, 2000.

[45] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96, New York, NY, USA, 2008. ACM.

[46] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[47] D. Thaler and C. Ravishankar. Using name-based mappings to increase hit rates. *Networking, IEEE/ACM Transactions on*, 6(1):1–14, Feb 1998.

[48] B. Tierney, *et al.* Distributed parallel data storage systems: a scalable approach to high speed image servers. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 399–405, New York, NY, USA, 1994. ACM.

[49] S. Vazhkudai, D. Thain, X. Ma, and V. Freeh. Positioning dynamic storage caches for transient data. *IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[50] C. Vecchiola, S. Pandey, and R. Buyya. High-performance cloud computing: A view of scientific applications. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:4–16, 2009.

[51] J. Weissman and S. Ramakrishnan. Using proxies to accelerate cloud applications. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[52] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.