Flexible Caches for Derived Scientific Data over Cloud Environments

David Chiu Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA
{chiud,agrawal}@cse.ohio-state.edu

Abstract

Scientific processes and analyses have become increasingly more data and compute intensive in the wake of the information sharing/mining age. Meanwhile, computing as a utility, that is, offering pay-as-you-go access to on demand supercomputing resources, has finally emerged in the form of Cloud computing. In this powerful, new model of computing, facilities for intelligent resource allocation must be supported in an effort to minimize utility cost. This paper focuses on the challenges of supporting efficient data retrieval and processing over a scientific data repository hosted in a cloud setting. We have developed a cooperative scheme for caching intermediate data results from scientific processes. This cache leverages the dynamic scalability of a cloud environment for reducing process execution times, and we propose resource-conservative algorithms for this purpose. A detailed evaluation of our system has also been performed, considering scalability, load balancing, and impact of dynamic scalability on diminishing execution time.

1 Introduction

Research in the general area of the data grid has proved beneficial to a number of users including data analysts and scientists. Such efforts have initiated novel methods for accessing compute and data intensive processes for deriving scientific data results. However, today's scientific simulations, observations, and experiments are capable of generating massive volumes of data per day. These data sets, due to their size, are typically stored in persistent mass storage systems, which provide expansive, but slow, disks across distributed systems, including the grid.

Much progress toward alleviating the hardships in large scientific computing have been realized in the grid workflow community. For instance, advancements in modeling semantics, metadata cataloging, and AI planning techniques provide automatic workflow synthesis by allowing relevant resources to be discovered correctly and efficiently [25, 26, 14, 6, 30]. Similarly, efforts toward optimizing distributed scheduling have ushered new heuristics in the context of heterogeneous networks [7, 32]. With mounds of other grid utilities brought into fruition for querying, deriving, and storing data [12, 3, 33], the challenges for provisioning efficient large scale scientific computing have become unsurprisingly limited by data movement and process execution. As a result, efficient access to multitudinous mass storage data repositories is becoming increasingly imperative in grid programming models.

Meanwhile, a powerful grid-*esque* computing model, known simply as the Cloud, has brought forth pragmatic provisions for *computing as a utility*. This model allows users on-demand access to supercomputing infrastructure — for a price. Evidenced by a surge of interest from multiple stakeholders in the industry and the academe, Cloud computing appears to be here to stay [2]. Providers, such as Amazon's Elastic Compute Cloud (EC2) [1] and Google's App Engine [15], have already made great strides toward offering this service to the mainstream. As they gain pace in the industry, users in many disciplines are afforded access to leverage any amount of computing power to satisfy their relative needs.

In a perspective astray from traditional distributed and grid paradigms, under the Cloud, computing resources should be scaled up or relaxed dynamically to optimize utility costs in handling varying workloads. This at-cost flexibility minimizes wasteful usage of allocated machines during times when their presence is superfluous for the task at hand. In terms of advancing systems for large scale scientific computations, the timely emergence of the Cloud can certainly be leveraged to help reduce overall execution times by a significant factor.

In this paper, we present a dynamic cooperative caching framework for storing reusable derived data, i.e., intermediate results. Our cache system, while currently implemented as a component in our scientific workflow management system [9, 10, 11], can be deployed as a general service and integrated transparently into frameworks. Our proposed cache provides an efficient indexing structure that captures domain-specific attributes of the cached intermediate results (in our system, these attributes are domain concepts and spatiotemporality). We also propose an algorithm to conservatively acquire Cloud nodes for caching on an as-needed basis. This approach seeks to minimize node allocation by assigning cached data greedily to already-acquired nodes, and therefore, reducing the utility costs of allocating compute nodes in the Cloud. Our cache utilizes both current database and web caching solutions to minimize indexing and migration costs respectively.

We have evaluated our system in terms of miss rate, node

utilization, and the added effects of the cache on reducing execution times. Our experimental results show that our flexible cache system is capable obtaining minimal miss rates while utilizing significantly less resources than statically allocated systems of fixed-sizes in the span of the system's runtime. We also show that the system scales to increasing workloads through its dynamic Cloud resource allocation scheme.

The rest of this paper is organized as follows. In the Section 2, we present background and goals for our cache framework. In Section 3, we formalize the structures and algorithms involved for our system. Experimental results are discussed in Section 4. We identify related works in Section 5 and conclude in Section 6.

2 System Goals and Background

Large scale applications in the data grid are typically modeled in a tertiary architecture involving (i) clients, (ii) middleware/grid services which access (iii) persistent data repositories in Mass Storage Systems (MSS). While many efforts have been made toward easing accesses to MSS through caching and file replication in between stages (ii) and (iii), in today's systems, scalability has become synonymous with performance. This is due to the importance for cutting edge applications to handle varying, unpredictable, and/or time-critical workloads. Whether the objective is load balancing, minimizing out-of-core executions, or reducing network traffic, the goal of scalable systems is singular: to provide high levels sustainability and availability.

Situations within certain applications, such as tropical storm tracking, flood monitoring, or even sudden transgressions in markets, often invoke increases in workloads due to heightened interests from various users. But because queries during these circumstances are often related, a considerable amount of redundancies among processes is present, and their intermediate and final results can often be reused to hasten subsequent queries. Our approach exploits these overlaps by caching and utilizing intermediate results when called upon at a later time. Our system is a distributed, cooperative cache which scales to increasing workloads by dynamically allocating Cloud nodes to share the work. But this simple solution is met with certain requirements and challenges.

2.1 System Goals

Provisioning Fast Access Methods

The ability to store large quantities of intermediate results is hardly useful without efficient access methods. These include not only identifying which node contains the cached data, but also facilitating fast hits and misses within that node. Clearly, the former goal could be achieved rather easily through such methods as hashing or directory services, but the latter requires more considerations toward indexing. Although the index structure is specific to certain implementations, we utilize spatiotemporal indices [18, 16, 4, 24] due to the wide range of applications that they can accommodate and also their de facto acceptance into most practical database systems.

Graceful Adaptation to Increasing Workloads

An increase in query 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, however, is imperative for facilitating fast response times in any cache system. The on-demand flexibility afforded by the Cloud is particularly important here – our system must be memory conscious and acquire nodes to share the storage burden, thus guaranteeing in-core access.

Conservative Utilization of Cloud Resources

Our cache framework is a form of utility computing over the Cloud, where resource allocation should be minimized, if possible, to control costs. This means that our system must (i) utilize a minimal amount of nodes for the respective workload and (ii) control the growth rate of nodes acquired. To facilitate these tasks, we propose a load distribution algorithm that is greedy in terms of its aggressive nature in assigning, and thus containing, the cache within the current system. Our algorithm, GBA (Greedy Bucket Allocation) acquires new nodes to handle increasing workflows as the last resort. In Section 3, we discuss technical details of this algorithm.

High-Level Integration with Existing Systems

Our cache must subscribe to an intuitive programming interface which allows for easy integration into existing systems. Like most caches, our system only presents high-level *search* and *update* methods while hiding internal nuances from the programmer, such as victimization and replacement policies, management of underlying resources, data movement, etc. In other words, our system can be viewed as a grid service, from the programmer's perspective, for caching intermediate results.

2.2 Previous Work

The work presented here is in the context of our previously developed geospatial workflow system [9, 10, 11], which enables automatic planning of composite services and data sets for answering high level queries. Workflow managers, historically rooted in managing large scale business processes, have resurfaced in the data grid as powerful scientific computing models. In these models, workflows may involve any number of process executions and data sets. For instance, consider the workflow, shown in Figure 1, which answers a query asking for shoreline derivation at some specified time and location, a typical query in the geospatial domain. This workflow involves the execution of four distinct data and compute intensive processes: (i) getStns, (ii) getWaterLevel, (iii) getCTM, (iv) getShoreline, which translates to four disparate opportunities for caching intermediate (and final) derived results. From the perspective of a workflow manager, the cache is accessed to locate intermediate results prior to the individual process invocations and updated after each time a result is obtained. Although specific to our system, the spatiotemporality of geospatial data sets presents

certain complexities in terms of indexing the intermediate results. We revisit this challenge in the next section.

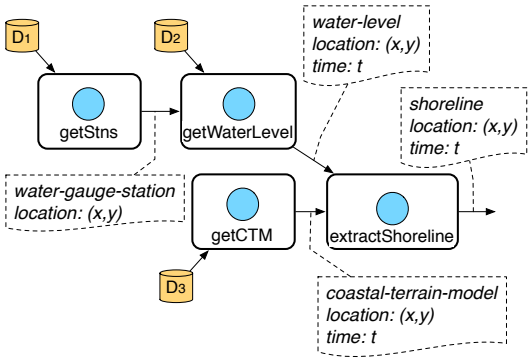

Figure 1. Shoreline Extraction Workflow

3 Cache Design and Access Methods

We now present the core components in our derived data cache scheme. In each subsection, we respectively address efficient data identification, node hashing, and cache access and maintenance methods in our framework.

3.1 Indexing Intermediate Results

To provide fast methods for identifying the existence of derived intermediate results in the cache, an effective index is required. The immediate challenge is deciding which attributes of the derived data are to be indexed. Earlier, we alluded to the fact that geospatial data are spatiotemporal, so the relevant *time* and *space* are natural indices. However, spatiotemporal indexing alone is inadequate; disparate phenomena can be of interest at the same location and time. Our geospatial workflow system employs a domain ontology which models the relationships of geospatial concepts and the available processes and data sets. In the example from Figure 1, the (process → concept) ontological mappings are: (getStns → water gauge station), (getWaterLevel → water level), (getCTM → coastal terrain model), and (getShoreline → shoreline). Thus, the index must also capture the data's domain concept along with its relative time and location of interest, e.g., any data that getShoreline produces should be indexed under the *shoreline* concept.

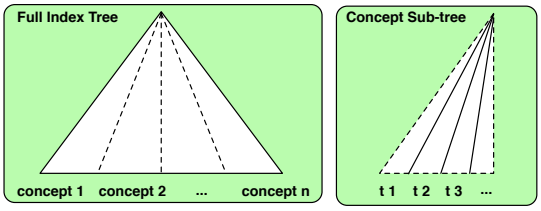

Figure 2. The Modified B^x-Tree

We utilize a modification of B^x-Trees [18] to index cached data. This particular index is useful in our approach due to its familiar and ubiquitous underlying structure, B+Trees[†]. Because B+Trees are widely accepted in most database systems, its integration is simplified. The modification to this structure that we propose is to allow its keys to also capture the data's ontological concepts. First, a linearization of domain concepts is obtained by mapping each to a distinct integer. This value is set to the leftmost binary portion of the key. Recall that B+Tree structures are balanced, and like binary trees, sorted incrementally on keys. Then, by attaching the concept mappings to the most significant bit portions of the key, we can logically partition the tree into independent concept sections. Each concept subtree is further partitioned into the times they represent, and finally, within each time partition lie the space filling curve (one dimensional) representations of spatial regions. This configuration is depicted in Figure 2. For an intermediate result pertaining to domain concept c is located in (x, y) with time t, its key is formulated as the bit string:

$$key(c, t, o) = [c]_2 \cdot [t]_2 \cdot [curve(x, y)]_2$$

where $curve(x, y)$ denotes the space filling curve mapping of (x, y), $[n]_2$ denotes the binary representation of n, and $\cdot$ denotes binary concatenation.

3.2 Consistent Hashing

The design of our caching scheme must be conducive to changes in its underlying structure. That is, adding and removing cache nodes should take minimal effort, which is a deceptively hard problem. To illustrate, consider an n-node cache system, where the node responsible for caching key k is computed $h(k) = (k \mod n)$, assuming that k denotes a key, computed via the aforementioned $key(c, t, o)$, used for indexing the accompanying data. Now assume that a new node is allocated, which effectively changes the hash function to $h'(k) = (k \mod n + 1)$. This ostensibly simple change forces all currently keyed records to be rehashed and relocated using $h'(k)$, otherwise requests for k would result in an unconditional miss due to inconsistencies between the new hash function and underlying structure. Rehashing and migrating large volumes of records after each node acquisition is, without saying, prohibitive for most systems.

To handle this problem, known as *hash disruption* [23], we implement consistent hashing [20], which has found applications primarily within distributed web proxies, among others. In this scheme, assume some fixed range of keys, $[0, r]$. Within this range exists a sequence of p buckets, $B = \{b_1, \ldots, b_p\}$, with each bucket mapped to a node. Figure 3 (top) shows an example of a cooperative cache system system based on consistent hashing that consists 2 nodes and 5 buckets. When a new key k arrives, it is first hashed via some function $h(k) \in [0, r]$ then assigned to the node pointed by $h(k)$'s closest upper bucket. In our figure, the incoming k is assigned to node n_2 via b_4. Often, but not necessary, the

[†]As per [18], B^x-Trees are, in the simplest sense, B+Trees whose keys are linearizations of spatiotemporal attributes via space filling curves and time partitioning.

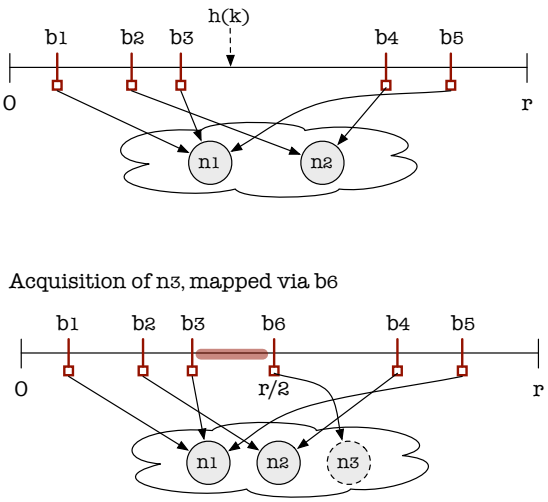

Figure 3. Consistent Hashing Example

hash line is implemented in a circular fashion, so in our example, a key $k \mid b_5 < h(k) \leq r$ would be mapped to n_1 via b_1.

Because bucket-to-node mappings are fixed, consistent hashing reduces hash disruption by a considerable factor. For instance, let us consider Figure 3 (bottom), where a new node, n_3, is acquired and assigned by bucket $b_6 = r/2$ to help share the load within the interval 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 complete rehash of all keys. Thus, consistent hashing simplifies the task of workload adaptation: First, if a node n becomes overloaded due to some large or popular bucket interval $(b_i, b_j]$, then we can alleviate the load on n simply by adding a new bucket, b_k in the crowded region of $(b_i, b_j]$ and assigning b_k to the least loaded node. Secondly, if all nodes within our system are full, then a new node from the Cloud can be quickly introduced without much effort, and its buckets can be assigned to crowded intervals to continue to help distribute the load evenly.

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 memory 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 $\forall_i : b_i \in [0, r]$ and $b_i < b_{i+1}$. Because consistent hashing inserts keys based on the upper bucket closest in proximity, an efficient implementation of the hash function, $h(k)$, can be achieved by running binary search for k over the ordered sequence, B, and returning the bucket at, or immediately larger than, k. In a circular implementation,

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

3.3 Cache Access Algorithms

Although the method for searching for some cached result is trivial, i.e., by running a B^x-Tree search for k on the node pointed by $h(k)$, the procedure for inserting into the cache is slightly more involved. In Algorithm 1, GBA (Greedy

Algorithm 1 GBA-insert(k, v)

```
1:  static NodeMap[...]
2:  b ← h(k)    /* hash k's bucket */
3:  n ← NodeMap[b]    /* get b's associated node */
4:  if ||n|| + sizeof(v) < ⌈n⌉ then
5:      n.insert(k, v)    /* insert directly on node n */
6:  else
7:      /* migrate keys (k', ..., k*) from node n to node n_min */
8:      K_b ← b.keys()    /* all keys w.r.t. bucket b */
9:      k' ← K_b.min()
10:     k* ← K_b.median()
11:     GBA-migrate(k', k*, n)
12:
13:     /* renew hash after migration changes structures */
14:     n ← NodeMap[h(k)]
15:     n.insert(k, v)
16: end if
```

Bucket Allocation) Insert is defined with a pair of inputs, k and v, denoting the key and value respectively. Here, v is a reference to the intermediate data. On Line 1, a hash map is brought into scope. This structure defines the relation $NodeMap[b] = n$ where n is the node mapped to bucket value b. After identifying k's bucket and node (Lines 2-3), the (k, v) pair is inserted into cache node n if the system determines that its insertion would not cause a memory overflow on n (Lines 4-5). Recalling the fact that caches expanding into disk memory would be prohibitively slow, if an overflow is detected, migration must be called to make space. The

Algorithm 2 GBA-migrate(k_{start}, k_{end}, n_{src})

```
1:  Ψ ← all pairs from n_src: {(k_ψ, v_ψ) | k_start ≤ k_ψ ≤ k_end}
2:  /* find minimally loaded node */
3:  n_min ← argmin_{n_i ∈ N} ||n_i||
4:  if ||n_min|| + sizeof(Ψ) > ⌈n_min⌉ then
5:      /* stolen keys and values will overflow n_min
6:              allocate new, empty node instead */
7:      n_min ← nodeAlloc()
8:  end if
9:  /* transfer stolen cache */
10: for all (k_ψ, v_ψ) ∈ Ψ do
11:     n.delete(k_ψ)
12:     n_min.insert(k_ψ, v_ψ)
13: end for
14: /* update structures */
15: static B = (...)
16: B ← (b_1, ..., b_i, k_end, b_{i+1}, ..., b_p) | b_i < k_end < b_{i+1}
17: NodeMap[k_end] ← n_min
```

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

the keys from the overflown bucket. The minimal and median key values, k' and k^*, relative to the overflown bucket, are found and used to call the migration algorithm, GBA-migrate (Lines 6-11). A quick rehash on k is needed in light of the modified B and $NodeMap[\ldots]$ structures incurred from running GBA-migrate (Line 14). The pair, (k, v) can finally be inserted into the rehashed node.

The GBA-migrate procedure, shown in Algorithm 2, inputs the range of keys to be migrated, k_{start} and k_{end}, and the node from which these keys should be *stolen*, n_{src}. The first step is to retrieve the set Ψ of all $(key, value)$ pairs from n_{src}. Next, the least loaded node, n_{min} is identified from the current cache configuration (Line 3). If Ψ cannot fit within n_{min}, however, a new node must be allocated from the Cloud (Lines 4-8). Lines 9-13 describe the transfer of Ψ from n to n_{min}. Finally, the bucket intervals, B, and $NodeMap[\ldots]$ structures are updated in the static scope (Lines 14-17). We

Algorithm 3 steal-key-pairs(k_{start}, k_{end})

```
1:  Ψ ← {}
2:  end ← false
3:  /* L is the leaf initially containing k_start */
4:  L ← bxTree.search(k_start)
5:  /* sweep leaf nodes until k_end */
6:  while (¬end ∧ L ≠ null) do
7:      /* each leaf node contains multiple keys */
8:      for all (k, v) ∈ L do
9:          if k ≤ k_end then
10:             Ψ ← Ψ ∪ {(k, v)}
11:         else
12:             end ← true
13:             break
14:         end if
15:     end for
16:     L ← L.nextLeaf()
17: end while
18: return Ψ
```

digress briefly to discuss the migration process, which was simplified in Algorithm 2. Line 1 of Algorithm 2 can be replaced with the more nuanced Algorithm 3. In order to *steal* all keys and their values between $[k_{start}, k_{end}]$, we perform a search-and-sweep on the B^x-Tree index. First, a search for k_{start} is invoked to locate its leaf node. Then, recalling that leaf nodes are arranged as a key-sorted linked list in B+-Trees, a sweep on the leaf level is run until k_{end} has been reached. This procedure is illustrated in Figure 4.

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, etc. To be succinct, we make the simple assumption that $sizeof((k, v)) = 1$ to normalize cached records, that is, $||n|| = n$.keys(). 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 analysis.

Since the algorithms are dependent, we begin with the analysis of Algorithm 3, whose time complexity is denoted

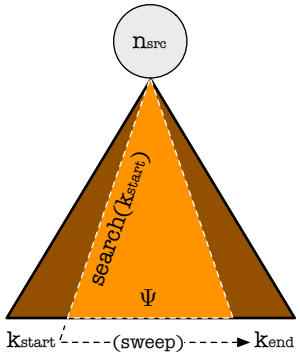

Figure 4. Migration on the B^x-Tree Index

T_{stl}. Because we are assuming the worst case, 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 due to our assumption of an even bucket/node distribution, which would cause Algorithm 1's calculation of k' and k^* to be assigned such that $k^* - k' \approx \lceil n \rceil / 2$, and thus T_{stl} can be analyzed as having an $O(\log_2 ||n||)$-time B^x-Tree search followed by a linear sweep of $\lceil n \rceil / 2$ records,

$$T_{stl} = \log_2 ||n|| + \lceil n \rceil / 2$$

This allows us to solve for $T_{migrate}$, the complexity of Algorithm 2, whose expansion involves T_{stl} followed by 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,

$$\begin{aligned} T_{migrate} &= T_{stl} + (\lceil n \rceil / 2) t_{net} \\ &= \log_2 ||n|| + \lceil n \rceil / 2 (t_{net} + 1) \end{aligned}$$

Finally, we are ready to solve for T_{GBA}, the run time 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 GBA-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},

$$T_{GBA} = \begin{cases} O(\log_2 p), & \text{if } ||n|| + 1 < \lceil n \rceil \\ O((\lceil n \rceil / 2) t_{net}), & \text{otherwise} \end{cases}$$

Although t_{net} is neither uniform nor trivial in practice, our analysis is still 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.

4 Experimental Results

In this section, we discuss the evaluation of our cooperative cache system. In our configuration, the workflow manager is executed on a Linux machine running Pentium IV 3.0Ghz Dual Core with 1GB of RAM, which is linked via a 10MBps connection to the Cloud. Our Cloud environment is emulated on a cluster, with each node having uniform bandwidths of 10MBps. In our experiments, all caches are initially cold.

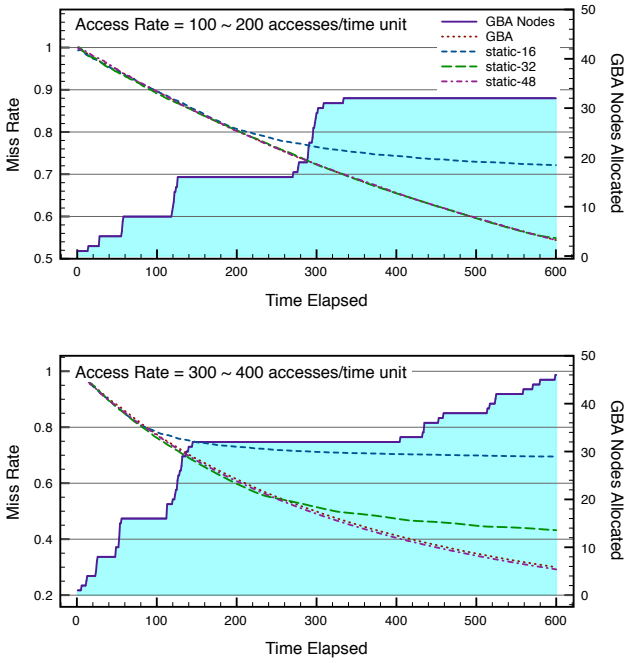

Figure 5. Behavior of Miss Rates

The first set of experiments measures miss rates over time to evaluate the adaptation of the cache with increasing workloads. For this experiment, we engage the following cache configurations: Three statically allocated caches consisting 16 (static-16), 32 (static-32), and 48 (static-48) cache nodes, compared against our cooperative scheme (GBA), which initially starts with one node. Let us focus on the top graph of Figure 5. For this plot, we produced varying workloads of 100 to 200 cache accesses per time unit, and ran it for 600 time units.[§] This graph clearly shows that the GBA's miss rates drop at the same rate as configurations containing 32 and 48 nodes. The miss rates for a 16 node configuration begins tapering off at 200 time units into the execution due to LRU victimization in order to keep its indices in-core. This problem does not affect the other three configurations, which implies that the workload is too light for 32 and 48 configurations. The graph also implies that the GBA-migration process is effective for adapting our cache to increasing workloads because GBA cache miss rates are equivalent to those provided by the larger 32 and 48 node configurations. The y-axis on the right hand side refers to node growth rate from running GBA. As we can see from the GBA Nodes trend, our system is able to provide ideal miss rates while using less nodes in the duration of the run, which saves on the Cloud's utility cost. Also note that GBA also allows us to use the minimal amount of nodes, 32, to achieve the same miss rate instead of overprovisioning: For static-32 and static-48, ideal miss rates are achieved, but both static configurations clearly utilize more computing resources during the run than GBA. This evidence supports that the GBA-based cache scheme will be financially cheaper to utilize over the Cloud's pay-as-you-go model.

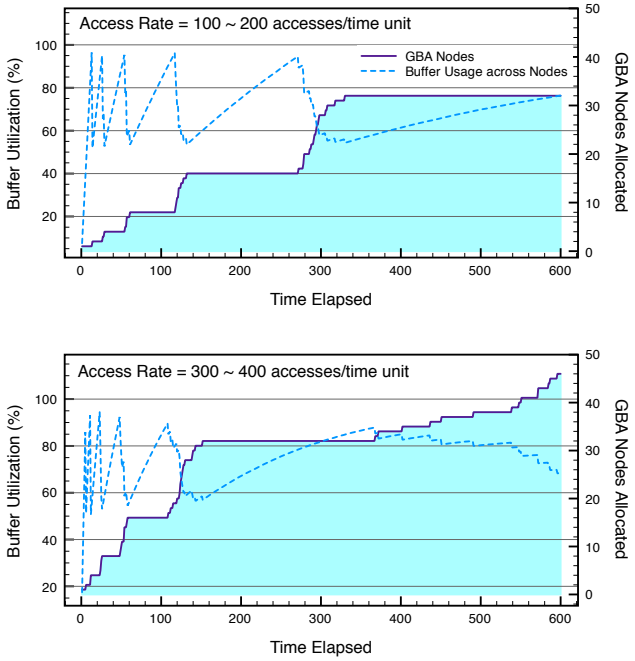

Figure 6. Balancing Memory Utilization

The natural extension to this experiment is to increase the workload. As we can see in the bottom graph of Figure 5, the first observation is that the miss rates improve significantly against the higher workload. This trend is expected, as cache systems thrive on usage. In this plot, the miss rates for 16 and 32 static node configurations level off due to victimization while GBA and the 48 node settings continue to improve until the end of execution, which is consistent with our expectations. Again, it is clear that GBA will provide better miss rates than static-16 and static-32 by utilizing more nodes on demand. However, it is still cheaper to use GBA over a static 48 node configuration albeit that static-48 provides equally ideal miss rates.

The next set of experiments offers insight into GBA's migration trends during these runs. The buffer utilization trend refers to average percentage of the buffer used on the cur-

[§] Although the unit of time is arbitrary in these experiments, larger units (e.g., minutes or hours) admittedly makes for a stronger case in using our cache for optimizing utility costs.

rently allocated nodes. Recall that the goal of the GBA-migrate algorithm is maintaining indices in-core by mapping additional buckets to lesser loaded nodes or by acquiring new nodes. In both graphs, the peaks and troughs in memory usage can be observed early in the execution due to the fast growth rates of the index against the low number of nodes allocated early. The greediness of our algorithm can be observed here, as evidenced by the usage spikes' closeness to 100% utilization right before new nodes are acquired to share the load. As expected, the usage trends flatten as more nodes are acquired throughout later stages of the execution. These graphs show that our system is capable keeping indices in-core dynamically through migration and that it scales to increasing workloads.

The final experiment displays the practical effects of our cache on workflow systems. We executed repeated runs of the scientific workflow for shoreline extraction seen earlier in Figure 1. The workflow, when executed without benefit of the cache, takes approximately 20 seconds to complete. We ran the workflow repeatedly, selecting random locations and times of relevance, and caching all results. As the main plot in Figure 7 shows, intermediate derived data caching improves the average workflow execution time over time. In fact, toward the end, GBA records a 5-time speed up over the non-cached version and significant speed ups over the static 8 node and 16 node configurations (due to victimization under these settings). Speed up over the 32 node configuration becomes evident toward the end of the run as GBA begins allocating more than 32 nodes at around 110×10^3 queries to avoid victimization.

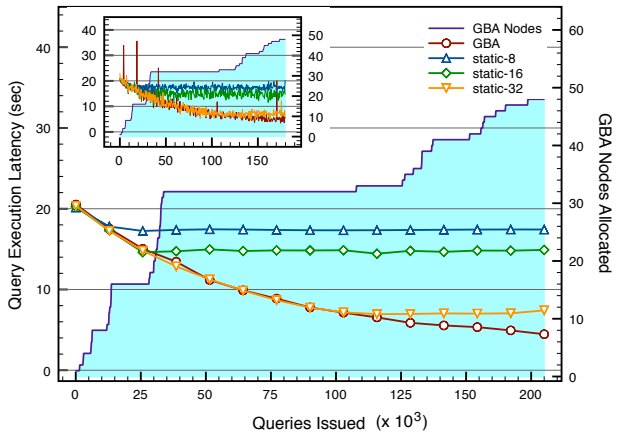

Figure 7. Cache Effects on Query Execution Time

The subgraph in this figure shows the costs of migration when running GBA, where the peaks in query latency correspond to heavier migration periods. But as the trend lines show, these heavy periods are amortized by the considerable amount of latency reductions over time. Furthermore, as we saw previously in Figure 6, the demand for migration also diminishes as execution proceeds, which allows us to attain the high speed up factors toward the end of the run.

5 Related Works

Although our system is novel in the sense of caching derived data on demand, it is much-inspired by efforts done in the general area of web caching [23]. To alleviate server load, intelligently placed proxies have historically been employed to replicate and cache popular web pages. The Internet Cache Protocol (ICP) is employed by many web proxy systems to exchange query messages [31], although our system does not currently implement this standard.

Several methods can be used to evenly distribute the load among cooperating caches. Gadde, Chase, and Rabinovich's CRISP proxy [13] 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 [20, 21] 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 [27]. Karger et al.'s technique is currently employed in our cooperative cache.

Efforts in storage management have introduced a layer of cache 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) [28] offer a buffer between clients and access to MSS and other storage systems including SDSC's Storage Resource Broker [3]. Other efforts, including works done by Otoo et al. [22], Bethel et al. [5], and Vazhkudai et al. [29], 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 [19]. Other grid utilities have sought for the storage of more detailed information on the scientific, such as virtual data traces, known as provenance. Chimera [12] is a system that stores information on virtual data sets which affords scientists the possibility to understand how, and why, certain data can be derived, as well as a way to reproduce data derivations. Virtual services [17], like our approach, stores service results in intermediate caches architectures, have also been briefly proposed. However, like all the aforementioned grid cache systems, it has not been considered under the context of the Cloud.

The goal of our framework seeks to expedite slow data accesses to large persistent storage by introducing a mediating cooperative cache system. This architecture is indeed tantamount to the above systems. However, our framework differs from those in its dynamic, Cloud-related aspects. Our proposed cache system is capable of growing to flexibly adapt to increasing workloads, which are prevalent in such compute-intensive environments as the data grid. Additionally, it is utility cost-conscious as to optimize the financial demands of Cloud users.

6 Concluding Remarks

Cloud providers have begun offering users at-cost access to high performance computing infrastructures. In this pa-

per, we propose a Cloud-based cooperative cache system for reducing execution times of scientific processes. The algorithms presented herein are cost-conscious as not to over-utilize Cloud resources. Performance evaluations show that our cooperative cache system is scalable to varying high workloads, cheaper than utilizing fixed networking structures on the Cloud, and effective for helping minimize execution times.

Although this paper focuses on conservative resource allocation, the Cloud model also needs consideration toward deallocation schemes for further reducing costs. The deallocation problem, largely ignored so far, can no longer be eluded in the context of the Cloud [2]. Although a number of literature exists for workload monitoring and prediction, our system may need heuristics for aggressive deallocation when a cost threshold is reached.

Acknowledgments

This work is supported by NSF grants 0541058, 0619041, and 0833101. The equipment used for the experiments reported here was purchased under the grant 0403342.

References

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

[2] 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.

[3] 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.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] 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.

[6] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *The 13th International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, Italy, 2003. AAAI.

[7] T. D. Braun, *et al.* A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[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. Enabling ad hoc queries over low-level scientific datasets. In *Proceedings of the 21th International Conference on Scientific and Statistical Database Management (SSDBM'09)*, 2009.

[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 IEEE/ACM International Conference on Grid Computing (Grid'08)*, 2008.

[12] 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.

[13] 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.

[14] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33, 2004.

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

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

[17] A. Jagatheesan, R. Moore, A. Rajasekar, and B. Zhu. Virtual services in data grids. *International Symposium on High-Performance Distributed Computing*, 0:420, 2002.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] H. Samet. The quadtree and related hierarchical structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[25] G. Singh, *et al*. A metadata catalog service for data intensive applications. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[26] B. Svrivastava and J. Koehler. Planning with workflows - an emerging paradigm for web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services*. ICAPS, 2004.

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

[28] 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.

[29] 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.

[30] T. Vossen, M. Ball, A. Lotem, and D. S. Nau. On the use of integer programming models in AI planning. In *IJCAI*, pages 304–309, 1999.

[31] D. Wessels and K. Claffy. Internet cache protocol (icp), version 2, 1997.

[32] M. Wieczorek, A. Hoheisel, and R. Prodan. Taxonomy of the multi-criteria grid workflow scheduling problem. In *CoreGrid Workshop*, 2007.

[33] Y. Zhao, *et al.* Virtual data grid middleware services for data-intensive science: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(6):595–608, 2006.