Evaluating Caching and Storage Options on the Amazon Web Services Cloud

David Chiu
School of Engineering and Computer Science
Washington State University
Vancouver, WA 98686

Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210

***Abstract*—With the availability of on-demand compute and storage infrastructures, many users are deploying data-intensive scientific applications onto Clouds. To accelerate these applications, the prospect of caching intermediate data using the Cloud's elastic compute and storage framework has been promising. To this end, we believe that an in-depth study of cache placement decisions over various Cloud storage options would be highly beneficial to a large class of users. While tangential analyses have recently been proposed, ours in contrast focuses on cost and performance tradeoffs of maintaining a data cache with varying parameters of any Cloud application. We have compared several Amazon Web Service (AWS) Cloud resources as possible cache placements: within machine instances (in-memory, on-disk, on large mountable disk volumes) and in cheap persistent Cloud storage (S3). We found that application-dependent attributes like unit-data size, total cache size, persistence requirements, etc., have far-reaching implications on the cost to sustain their cache. Also, while instance-based caches expectedly yield higher cost, the performance that they afford may outweigh lower cost options.**

***Keywords*-Cloud storage and caching strategies, cost analysis**

I. Introduction

The mounting growth of scientific data has spurred the need to facilitate highly responsive compute- and data-intensive processes. Such large-scale applications have traditionally been hosted on commodity clusters or grid platforms. However, the recent emergence of on-demand computing is causing many to rethink whether it would be more cost-effective to move their projects onto the Cloud. Several attractive features offered by Cloud providers, after all, suit scientific applications nicely. Among those, elastic resource provisioning enables applications to expand and relax computing instances as needed to scale and save costs respectively. Affordable and reliable persistent storage are also amenable to supporting the data deluge that is often present in these applications.

A key novel consideration in Cloud computing is the pricing of each resource and the resulting costs for the execution of an application. Together with considerations like wall-clock completion time, throughput, scalability, and efficiency, which have been metrics in traditional HPC environments, the cost of execution of an application is very important. Several recent studies have evaluated the use of Cloud computing for scientific applications with this consideration. For example, Deelman, *et al.* studied the practicality of utilizing the Amazon Cloud for an astronomy application, Montage [9]. Elsewhere, researchers discussed the challenges in mapping a remote sensing *pipeline* onto Microsoft Azure [13]. In [12], the authors studied the cost and feasibility of supporting BOINC [2] applications, e.g., SETI@home, using Amazon's cost model. Vecchiola, *et al.* deployed medical imaging application onto Aneka, their Cloud middleware [17]. An analysis on using storage Clouds [14] for large-scale projects have also been performed. While other such efforts exist, the aforementioned studies are certainly representative of the growing interest in Cloud-supported frameworks. However, there are several dimensions to the performance and cost of executing an application in a Cloud environment. While CPU and network transfer costs for executing scientific workflows and processes have been evaluated in these efforts, several aspects of the use of Cloud environments require careful examination.

In this paper, we focus on evaluating the performance and costs associated with a number of caching and storage options offered by the Cloud. The motivation for our work is that, in compute- and data-intensive applications, there could be considerable advantage in caching intermediate data sets and results for sharing or reuse. Especially in scientific workflow applications, where task dependencies are abundant, there could be significant amounts of redundancy among related processes [19], [6]. Clearly, such tasks could benefit from fetching and reusing any stored precomputed data. But whereas the Cloud offers ample flexibility in provisioning the resources to store such data, weighing the tradeoff between performance and usage costs makes for a compelling challenge.

The Amazon Web Service (AWS) Cloud [4], which is being considered in this paper, offers many ways for users to support such a cache. In one approach, a cohort of virtual machine instances can be allocated, and data can be stored either on disk or in memory (for faster access, but with limited capacity). The costs of maintaining such a cache would also be much higher, as users are charged a fixed rate per hour. This fixed rate is moreover dependent on the requested machine instances' processing power, memory capacity, bandwidth, etc. On the other hand, AWS's Simple Storage Service (S3) can also be used store data. It could be a much cheaper alternative, as users are charged a fixed rate per GB stored per month. Data are also persisted on S3, but because of this overhead, we might expect some I/O delays. However, depending on the application user's requirements, performance may well outweigh costs or vice versa.

We offer an in-depth view of these tradeoffs in employing various AWS options for caching data to accelerate data-intensive processes, and our contributions are as follows. We evaluate performance and cost behavior given various average data sizes of an application. Several combinations of Cloud features are evaluated *vis à vis* as possible cache storage options. We furthermore consider several requirements, including data persistence requirements, cost, and high-performance needs. We believe that our analysis would be useful to the computing community by offering new insights into employing the AWS Cloud. Our experimental results may also generate ideas for novel cost-effective caching strategies.

The remainder of this paper is organized as follows. In Section II, we present some background into the AWS Cloud services, their cost model, and we briefly explain a Cloud-enabled cache framework we had previously developed for supporting large-scale computing environments. We present the results from detailed experiments in Section III. Related works are discussed in Section IV, and we conclude and briefly discuss future plans in Section V.

II. Background

In this section, we initially present a general cache framework, and then briefly present the various Infrastructure-as-a-Service (IaaS) features offered by the Amazon Web Services (AWS) Cloud, which includes persistent storage and on-demand compute nodes.

A. Cache Framework

Large-scale applications such as scientific workflows, data analysis, simulation, etc., are traditionally highly data-intensive. This characteristic is responsible for long-running compute processes, which is further compounded by data movement between compute nodes. For example, within scientific workflow applications, processes may have certain dependencies on each other's execution. These dependencies may require data to be transferred to the next set of processes before they can execute or be combined to generate some result. Such data-intensive situations are not singular to workflow applications, and data movement and computation are clearly bottlenecks. To accelerate these types of processes, caches can be deployed to maintain some set of precomputed/intermediate data for reuse. Especially in scientific applications, precomputed data could not only replace the need to compute redundant information, but it can also significantly reduce the amount of data transfers required.

For this purpose, we have previously proposed a cooperative cache that had been deployed onto the grid for reducing the overall processing times of scientific workflows [6]. We have since developed a self-managed variant of this cache framework, which automatically allocates machine instances as to expand the cache's capacity, as well as a heuristic to relax instances to save costs [7]. Thus, each node is responsible for a fraction of the entire data cache, which is managed using a consistent hashing approach [16], seen in Figure 1. In this figure, a hash line running from 0 to $r-1$ is configured. Each data set is first hashed onto this line using $h'(k) = (k \bmod r)$. The node which should store data set, k, is then referenced by the nearest *bucket* larger than k.

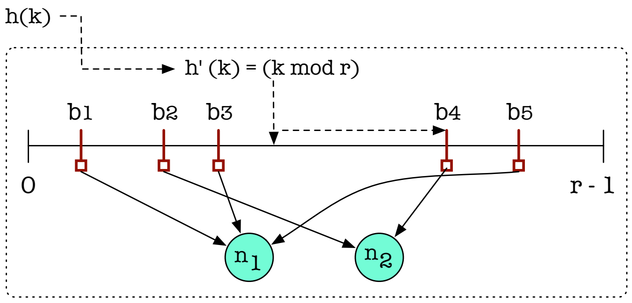


Fig. 1. Cooperating Cache Containing Two Nodes

A consistent hashing scheme allows minimal disruption to our cooperating system as we add and remove nodes to our cache dynamically. Indeed, since the hash function is fixed, only a small amount of data must be moved from one node to another when a node is introduced or relinquished into the scheme. Each node further contains B+-Tree which indexes the stored data sets. Our cache provides a high-level API, and can be accessed as a service so it is also non-intrusive to existing systems. Detailed descriptions can be found in [7].

Depending on the needs of an application, a high-performance cache can be deployed over Cloud nodes that have large amounts of memory and high bandwidth, at a higher cost. Similarly, persistent Cloud storage services or smaller Cloud nodes may also be used as less costly option. We envision a cache manager that can select and manage the best such configuration given some cost and QoS constraints. To do this, we must first evaluate the cost and performance of these resource configurations.

B. Cloud Services and Costs

AWS offers many options for on-demand computing as a part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities.

TABLE I
Amazon Web Services Costs

AWS Feature	**Cost (USD)**
S3	\$0.15 per GB-month \$0.15 per GB-out \$0.01 per 1000 in-requests \$0.01 per 10000 out-requests
Small EC2 Instance	\$0.085 per allocated-hour \$0.15 per GB-out
Extra Large EC2 Instance	\$0.68 per allocated-hour \$0.15 per GB-out
EBS	\$0.10 per GB-month \$0.10 per 1 million I/O requests

For example, a Small EC2 Instance (`m1.small`), according to AWS[1] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. AWS also states that the Small Instance has *moderate* network I/O. Another instance type we consider is the Extra Large EC2 instance (`m1.xlarge`), which contains 15 GB memory, 4 virtual cores with 2 EC2 Compute Units each, 1.69 TB disk storage with *high* I/O Performance. Their costs are shown in Table I. We focus on these two highly contrasting instance types in this paper because they offer a wide spectrum between capabilities, while also noting that several other instance types are in fact available in EC2.

Amazon's persistent storage framework, Simple Storage Service (S3), provides a key-value store with simple ftp-style API: `put`, `get`, `del`, etc. Typically, the unique keys are represented by a filename, and the values are themselves the data objects, i.e., files. While the objects themselves are limited to 5 GB, the number of objects that can be stored in S3 is unlimited. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very cheap (see Table I) to store data on S3.

Another option for persistent storage is to employ Elastic Block Storage (EBS) in conjunction with EC2 instances. The EBS service is a persistent disk volume that can be mounted directly onto a running EC2 instance. The size of an EBS volume is user defined and limited to 1 TB. Although an EBS volume can only be attached to one instance at any time, an instance can conversely mount multiple EBS volumes. From the viewpoint of the EC2 instance, the EBS volume can be treated simply as a local filesystem.

C. Tradeoffs

We now offer a brief discussion on the tradeoffs of deploying our cache over the aforementioned Cloud resources.

In-Instance-Core Option: There are several advantages in supporting our cache over EC2 nodes in terms of flexibility and throughput. Depending on the application, it may be possible to store all cached data completely directly in memory, which reduces access time. But because small instances consist only 1.7 GB of memory, we may need to dynamically allocate more instances to cooperate in establishing a larger capacity. On the other hand, we could also allocate an extra large instance with much more memory capacity. However, the instance could overfit our cache needs, which would betray cost-effectiveness. Because of these reasons, we would expect a memory-based cache to be the most expensive, but possibly with the highest performance, especially for an abundance smaller data units.

In-Instance-Disk Option: In cases where larger amounts of data are expected to be cached, we could store on the instance's disk. Even small EC2 instances provide ample disk space (160 GB), which would save us from having to allocate new instances very frequently for capacity, as we would expect in the previous option. However, disk accesses could be very slow compared to an in-memory cache if request rates are high. Conversely, if the average data size is large, disk access overheads may be amortized over time. We can expect that this disk-based option should be cheaper than the memory-based, with slightly lower performance depending on the average unit-data size.

[1]AWS Instance Types, http://aws.amazon.com/ec2/instance-types

Persistent Options: The previous two configurations do not account for persisting data. That is, upon node failure, all data is presumed lost even if stored on disk. Moreover, it can be useful to stop and restart a cache, perhaps during peak/non-peak times, to save usage costs.

The simplest persistent method is to directly utilize S3 to store cached data. This avoids any indexing logic from the application developer, as we can subscribe to S3's API. It is very inexpensive to store data on S3 and more importantly, because S3 is independent from EC2, we further elude instance allocation costs. However, due to S3's reliability and availability guarantees, it implements an architecture which supports replication and consistency, which would likely impact performance. Also, although storage costs are low, the data transfer costs are equivalent to those of EC2 instances, which leads to the expectation that high-throughput environments may not benefit cost-wise from S3.

Another persistent method are EBS volumes. One difference between EBS and S3 is that EBS volumes are less accessible. They must first be mounted onto an EC2 instance. But because they are mounted, it alludes to the potential for higher throughput than S3, whose communications is only enabled through high-level SOAP/REST protocols that ride over HTTP. Also in contrast to S3, EBS volumes are not unlimited in storage, and their size must be predefined by users. In terms of cost, however, EBS invokes a storage and request overhead to the hourly-based EC2 instance allocation costs. In the next section we evaluate these options in depth.

III. Experimental Results

We now discuss the evaluation of various cache and storage schemes over the aforementioned AWS Cloud resources

A. Experimental Setup

Resource Configuration: We have run experiments over the following configurations:

- `S3`: Data stored as files directly onto the S3 storage service (persistent).
- `ec2-m1.small-mem`: Data stored in memory on Small EC2 instance (volatile, moderate I/O).
- `ec2-m1.small-disk`: Data stored as files on disk on Small EC2 instance (volatile, moderate I/O).
- `ec2-m1.small-ebs`: Data stored as files on a mounted Elastic Block Store volume on small EC2 instance (persistent, moderate I/O).
- `ec2-m1.xlarge-mem`: Data stored in memory on Extra Large EC2 instance (volatile, high I/O).
- `ec2-m1.xlarge-disk`: Data stored as files on disk on Extra Large EC2 instance (volatile, high I/O).

- `ec2-m1.xlarge-ebs`: Data stored as files on a mounted Elastic Block Store volume on Extra Large EC2 instance (persistent, high I/O).

Within the EC2 instances, we evaluate three disparate ways to store the cache: in-core (`*-mem`), on local disk (`*-disk`), and on Amazon's Elastic Block Storage, or simply EBS (`*-ebs`). In both `m1.small` (32-bit) and `m1.xlarge` (64-bit) systems, we employ the Ubuntu Linux 9.10 Server image provided by Alestic.[2]

Application: As a representative workload, we performed repeated execution on a *Land Elevation Change* process, 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. This process inputs a triple, (L, t, t'), where location, L, denotes a coordinate and $t, t' : t < t'$ represent the times of interest. The service locates two Digital Elevation Model (DEM) files with respect to (L, t) and (L, t'). DEMs are represented by large matrices of an area, where each point denotes an elevation reading. Next, the process takes the difference between the two DEMs, which derives a DEM of the same size, where each point now represents the change in elevation from t to t'. We do stress that, while all our experiments were conducted with this process, by changing certain parameters, we can capture a variety of applications.

We have fixed the parameters of this process to take in 500 possible input keys, i.e., 500 distinct (L, t, t') triples, and our experiment queries randomly over this range. These input keys represent linearized coordinates and date. 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 process.

TABLE II
BASELINE EXECUTION TIME FOR LAND ELEVATION CHANGE PROCESS

DEM Size	Execution Time (sec)
1 KB	2.09
1 MB	6.32
5 MB	20.52
50 MB	75.39

To analyze cache and storage performance, which can be affected by memory size, disk speed, network bandwidth, etc., we varied the sizes of DEM files: 1 KB, 1 MB, 5 MB, 50 MB. One such file is output from one execution, and over time, we would need to store a series of such files in our cache. These sizes allow for scenarios from cases where all cached data can fit into memory (e.g., 1 KB, 1 MB) to cases where in-core containment would be infeasible (e.g., 50 MB), coercing the need for disk or S3 storage. The larger data will also amortize network latency and overheads, which increases throughput. The baseline execution times of the service execution are summarized for these input sizes in Table II.

[2]Alestic, http://alestic.com/

B. Performance Evaluation

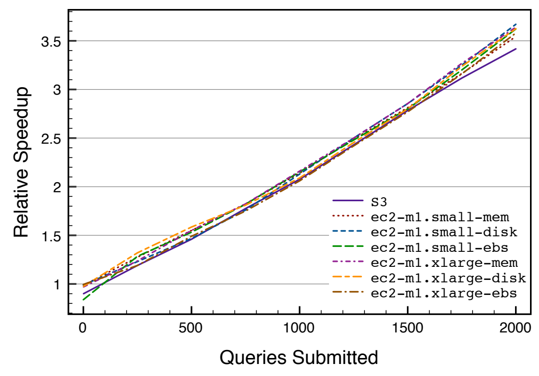
In this subsection, we measure performance in terms of relative speedup to the baseline execution times shown in Table II, as well as the overall throughput of the various system configurations. We randomly submitted 2000 queries over the 500 possible (L, t, t') inputs to the *Land Elevation Change* process. The querying strategy is not unlike most cache-aware systems. Each query submission first checks our cache, and on a miss, the process is executed and its derived result is transferred and stored in the cache for future reuse. Upon a hit, the result is transmitted immediately to the user. To ensure consistency across all experiments, we do not consider cache eviction here, and all caches start cold.

Relative Speedup: Let us first consider the relative speedup for 1 KB, 1 MB, 5 MB, and 50 MB DEM files, shown in Figure 2(a), 2(b), 2(c), and 2(d) respectively.

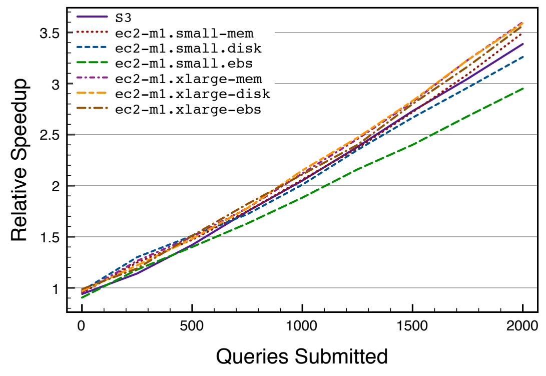
In Figure 2(a), it is somewhat surprising that the same speedup is observed in the across all configurations. Because the data size is small, we believe that this is due to internal memory caching mechanisms within the machine instances. Also, the network transmission time for 1 KB data is so insignificant as to not favor either *moderate* or *high* I/O. `S3`'s performance appears to drop relative to the instance-based settings toward the end of the experiment. Due to `S3`'s persistence features, this was not completely unexpected, and we posit that larger files may amortize `S3` invocation overhead. This becomes clear when we evaluate the average hit times per configuration later in this section.

The results for 1 MB and 5 MB DEMs are shown in Figures 2(b) and 2(c) respectively, with one caveat: The `ec2-m1.small-mem` configuration cannot fit all the data completely in its memory in the case of 5 MB. Fortunately, our cache had been designed to handle such cases (recall from Section II that our cache can add/remove cooperating nodes as needed. Every time a memory overflow is imminent, we allocate a new `m1.small` instance and migrate half the records from the overflown node to the new instance. In Figure 2(c), each instance allocation is marked as a triangle on `ec2-m1.small-mem`. Instance allocation is no doubt responsible for the performance slowdown.

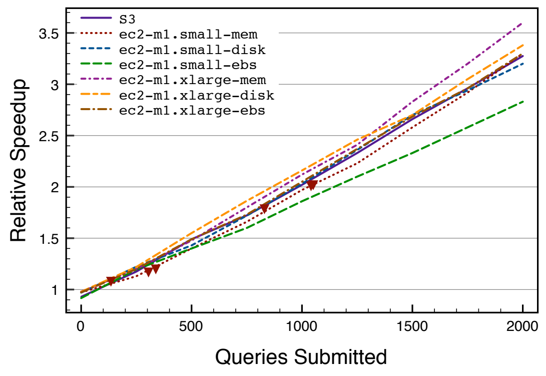
In Figure 2(b), it becomes clear that in-memory containment is, in fact, beneficial for both small and extra large instance types. However, the *high* I/O afforded by the `m1.xlarge` instance marks the difference between the two memory-bound instances. This is justified by the fact that the persistent `ec2-m1.xlarge-ebs` eventually overcomes `ec2-m1.small-mem`. Also interesting is the performance degradation of the small disk-bound instances, `ec2-m1.small-disk` and `ec2-m1.small-ebs`, which performs comparably to `S3` during the first $\sim$ 500 queries. Afterward, their performance decreases below `S3`. This is an interesting observation, considering that the first 500 queries are mostly cache misses (recall we start all caches out cold), which implies that the `m1.small` disk-based instance re-



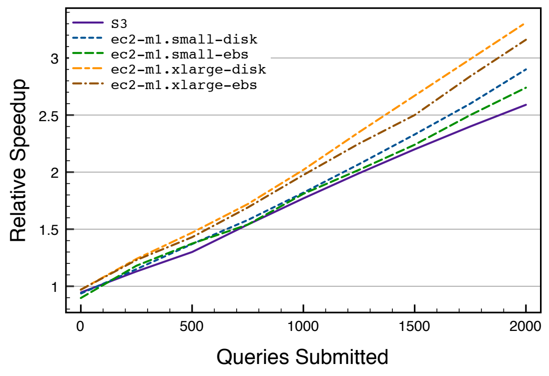
(a) Data Size = 1 KB



(b) Data Size = 1 MB

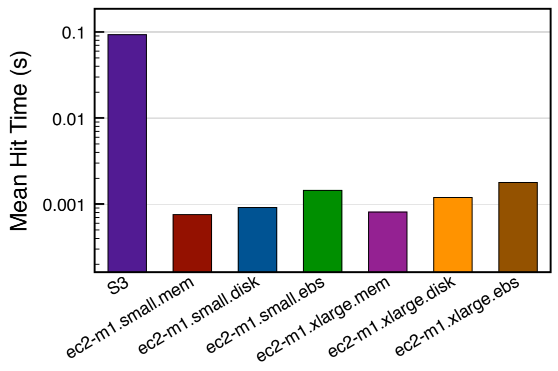


(c) Data Size = 5 MB

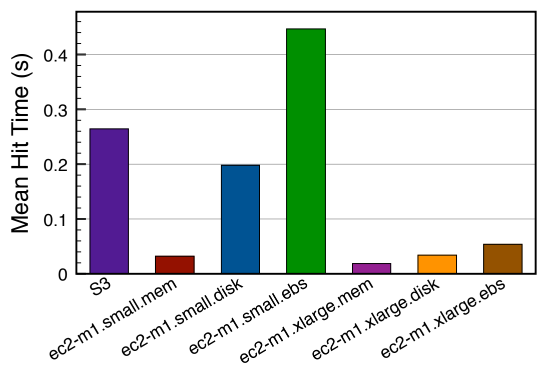


(d) Data Size = 50 MB

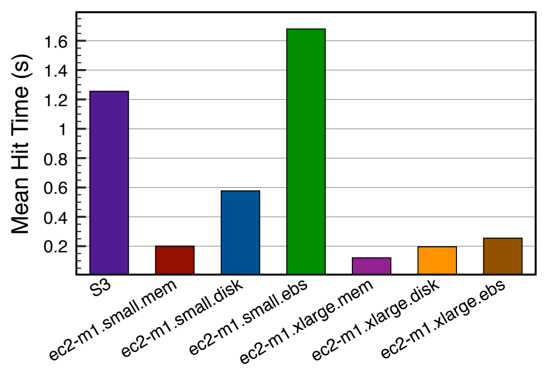
Fig. 2. Relative Speedup



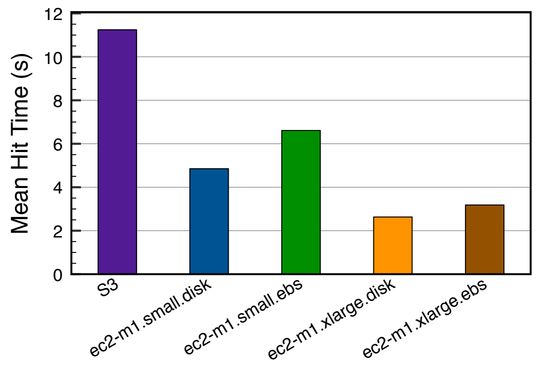
(a) Data Size = 1 KB



(b) Data Size = 1 MB



(c) Data Size = 5 MB



(d) Data Size = 50 MB

Fig. 3. Mean Cache Hit + Retrieval Time

trieves and writes the 1 MB files to disk faster than `S3`. However, when queries start hitting the cache more often after the first 500 queries, the dropoff in performance indicate that repeated random disk reads on the `m1.small` instances are generating significant overhead, especially in the case of the persistent `ec2-m1.small-ebs`.

Similar behavior can be observed for the 5 MB case, shown in Figure 2(c). The overhead of node allocation for `ec2-m1.small-mem` is solely responsible for its reduction in speedup. While the results are as expected, we do concede that our system's conservative instantiation of seven `m1.small` instances (that is, 1 to start + 6 over time) to hold a total of 500×5 MB of data in memory is indeed an overkill. Our instance allocation method was conservative here to protect against throttling, that is, the possibility that an instance becomes overloaded and automatically stores on disk. Clearly, these cases would invalidate speedup calculations.

Finally, we experimented with 50 MB DEM data files. As a representative size for even larger sized data often seen in data analytic and scientific processes, we operated under the assumption that memory-bound caching would most likely be infeasible, and we experimented only with disk-bound settings. One interesting trend is the resurgence of `ec2-m1.small-disk` and `ec2-m1.small-ebs` over `S3`. One explanation may be that disk-bound caches favor larger files, as it would amortize random access latency. It may also be due to `S3`'s persistence guarantees – we noticed, on several occasions, that `S3` prompted for retransmissions of these larger files.

Cache Access Time: In all of the above experiments, the difference among speedups still seem rather trivial, albeit some separation can be seen toward the end of most experiments. We posit that, had the experiments run much longer (i.e., much more than only 2000 requests) the speedups will diverge greatly. To justify this, Figures 3(a), 3(b), 3(c), and 3(d) show the average *hit* times for each cache configuration. That is, we randomly submitted queries to full caches, which guarantees a hit on every query, and we are reporting the mean time in seconds to search the cache and retrieve the relevant file.

Here, the separation among the resource configuration becomes much clearer. Figure 3(a) shows that using `S3` for small files eventually exhibits slowdowns by 2 orders of magnitude. This fact eluded our observation previously in Figure 2(a) because the penalty caused by the earlier cache misses dominated the overall times. In the other figures, we again see justification for using memory-bound configurations, as they exhibit for the lowest mean hit times. Also, we observe consistent slowdowns for `ec2-m1.small-disk` and `ec2-m1.small-ebs` below `S3` in the 1 MB and 5 MB cases. Finally, using the results from Figure 3(d), we can conclude that these results again support our belief that disk-bound configurations of the small instance types should be avoided for such mid-sized data files due to disk access latency. Similarly for larger files, `S3` should be avoided in favor of `ec2-m1.xlarge-ebs` if persistence is desirable. We have also ascertained from these experiments that the *high* I/O that is promised by the extra large instances contributes significantly to the performance of our cache.

C. Cost Evaluation

In this subsection, we present an analysis on cost for the instance configurations considered. The costs of the AWS features evaluated in our experiments are summarized in Table I. While in-Cloud network I/O is currently free, in practice, we cannot assume that all users will be able to compute within the same Cloud. We thus assume that cache data is transferred outside of the AWS Cloud network in our analysis. We are repeating the settings from the previous set of experiments, so an average unit data size of 50 MB will yield a total cache size of 25 GB of Cloud storage (recall that there are 500 distinct request keys). We are furthermore assuming a fixed rate of $R = 2000$ requests per month from clients outside the Cloud. We have also extrapolated the costs (right side of table) for when request rate $R = 200000$, using the Mean Hit Times from Figure 3 as the limits for such a large request rate R. Clearly, as R increases for a full cache, the speedup given by cache will eventually become denominated by the Mean Hit Times.

The cost, C, of maintaining our cache, the speedup S (after 2000 and 200000 requests), and the ratio C/S (i.e., the cost per unit-speedup), are reported under two requirements: volatile and persistent data stores. Again, volatile caches are less reliable in that, upon a node failure, all data is lost. The costs for sustaining a volatile cache for one month is reported in Table III. Here, the total cost can be computed as $C = (C_{Alloc} + C_{IO})$, where $C_{Alloc} = h \times k \times c_t$ denotes hours, h to allocate some k number of nodes using the said costs (from Table I) for instance type t. $C_{IO} = R \times d \times c_{io}$ accounts for transfer costs, where R transfers were made per month, each involving d GB of data per transfer, multiplied by the cost to transfer per GB, c_{io}.

First, we recall that if the unit-data size, d, is very small (1 KB), we can obtain excellent performance for any volatile configuration. This is because everything easily fits in memory, and we speculate that, even for the disk-based options, the virtual instance is performing its own memory-based caching, which explains why performance is not lost. This is further supported by the speedup when $d = 1$ MB, under the disk-based option. When projected to $R = 200000$ requests, we observe lucrative speedups, which is not surprising, considering the fast access and retrieval times for such a small file. Furthermore, when $R = 2000$ requests, the `ec-m1.small-disk` option offers excellent C/S ratios, making it a very good option. Conversely, when request rate R is large, the I/O performance of the small instances accounts for too much of a slowdown, resulting in low speedups, and a low C/S ratio. This suggests that `m1.xlarge` is a better option for systems expecting higher throughput rates.

Next, we compiled the cost for persistent caches, supported by `S3` and `EBS` in Table IV. Here, C_S refers to the cost per

TABLE III
MONTHLY VOLATILE CACHE SUBSISTENCE COSTS (S = SPEEDUP, C/S = COST PER UNIT-SPEEDUP)

		2000 Requests			200000 Requests		
Unit-Size		S	$C = C_{Alloc} + C_{IO}$	C/S	S	C_{IO}	C/S
1 KB (500 KB total)	`m1.small-mem`	3.54	$\$63.24 = \$63.24 + \$0.00$	\$17.84	2629.52	\$0.03	\$0.03
	`m1.small-disk`	3.67	$\$63.24 + \$0.00 = \$63.24$	\$17.23	2147.681	\$0.03	\$0.03
	`m1.xlarge-mem`	3.64	$\$505.92 = \$505.92 + \$0.00$	\$138.99	2302.43	\$0.03	\$0.22
	`m1.xlarge-disk`	3.63	$\$505.92 = \$505.92 + \$0.00$	\$139.57	1823.215	\$0.03	\$0.28
1 MB (500 MB total)	`m1.small-mem`	3.5	$\$63.54 = \$63.24 + \$0.30$	\$18.17	267.19	\$30.00	\$0.35
	`m1.small-disk`	3.26	$\$63.54 = \$63.24 + \$0.30$	\$19.49	28	\$30.00	\$4.06
	`m1.xlarge-mem`	3.6	$\$506.22 = \$505.92 + \$0.30$	\$140.62	347.3	\$30.00	\$1.53
	`m1.xlarge-disk`	3.59	$\$505.92 + \$0.30 = \$506.22$	\$141.13	180.53	\$30.00	\$2.94
5 MB (2.5 GB total)	`m1.small-mem`	3.3	$\$444.18 = \$442.68 + \$1.50$	\$96.27	109.47	\$150.00	\$4.26
	`m1.small-disk`	3.2	$\$64.74 = \$63.24 + \$1.50$	\$20.24	33.84	\$150.00	\$6.30
	`m1.xlarge-mem`	3.6	$\$506.78 = \$505.92 + \$1.50$	\$140.78	174.42	\$150.00	\$3.76
	`m1.xlarge-disk`	3.38	$\$506.78 = \$505.92 + \$1.50$	\$149.94	111.71	\$150.00	\$5.87
50 MB (25 GB total)	`m1.small-disk`	2.9	$\$78.74 = \$63.24 + \$15.00$	\$27.16	16.05	\$1500.00	\$97.40
	`m1.xlarge-disk`	3.31	$\$520.92 = \$505.92 + \$15.00$	\$152.85	31.66	\$1500.00	\$63.36

TABLE IV
MONTHLY PERSISTENT CACHE SUBSISTENCE COSTS (S = SPEEDUP, C/S = COST PER UNIT-SPEEDUP)

		2000 Requests			200000 Requests		
Unit-Size		S	$C_{S3} = C_S + C_R + C_{IO}$ $C_{EBS} = C_{Alloc} + C_S + C_R + C_{IO}$	C/S	S	C_{IO}	C/S
1 KB (500 KB total)	`S3`	3.4	$\$0.0023 = \$0.00 + \$0.002 + \$0.0003$	\$0.0007	24.79	\$0.23	\$0.01
	`m1.small-ebs`	3.62	$\$63.49 = \$63.24 + \$0.0002 + \$0.00 + \$0.0003$	\$17.54	1984.5	\$0.48	\$0.04
	`m1.xlarge-ebs`	3.58	$\$506.17 = \$505.92 + \$0.0002 + \$0.00 + \$0.0003$	\$141.39	2091.7	\$0.48	\$0.25
1 MB (500 MB total)	`S3`	3.39	$\$0.38 = \$0.075 + \$0.002 + \$0.30$	\$0.12	29.98	\$30.01	\$1.01
	`m1.small-ebs`	2.95	$\$63.59 = \$63.24 + \$0.05 + \$0.00 + \$0.30$	\$21.56	13.62	\$30.25	\$6.87
	`m1.xlarge-ebs`	3.57	$\$506.27 = \$505.92 + \$0.05 + \$0.00 + \$0.30$	\$142.00	133.96	\$30.25	\$4.01
5 MB (2.5 GB total)	`S3`	3.27	$\$1.88 = \$0.375 + \$0.002 + \$1.50$	\$0.58	19.97	\$150.00	\$7.53
	`m1.small-ebs`	2.83	$\$64.99 = \$63.24 + \$0.25 + \$0.00 + \$1.50$	\$22.97	11.84	\$150.27	\$18.04
	`m1.xlarge-ebs`	3.3	$\$507.67 = \$505.92 + \$0.25 + \$0.00 + \$1.50$	\$153.92	74.66	\$150.27	\$8.79
50 MB (25 GB total)	`S3`	2.59	$\$18.75 = \$3.75 + \$0.002 + \$15.00$	\$7.24	6.43	\$1500.00	\$233.87
	`m1.small-ebs`	2.74	$\$80.74 = \$63.24 + \$2.50 + \$0.00 + \$15.00$	\$29.47	11.09	\$1502.52	\$142.70
	`m1.xlarge-ebs`	3.16	$\$520.42 = \$505.92 + \$2.50 + \$0.00 + \$15.00$	\$164.69	22.66	\$1502.52	\$88.63

1 GB-month storage, C_R is the request cost, and C_{IO} refers to the data transfer costs per GB transferred out. Initially, we were surprised to see that `S3`'s C/S ratio is comparable to `EBS` (and even to the volatile options) when request rate R is low regardless of data size. However, for a large request rate, R, its overhead begins to slowdown its performance significantly compared to `EBS` options. Especially observed when unit-size, d, is very small, `S3`'s speedup simply pales in comparison to other options. Its performance expectedly increases as d becomes larger, due to the amortization of overheads when moving larger files. This performance gain of `S3`, however, drops sharply when $d = 50$ MB, resulting in only 6.43$\times$ speedup, making `EBS` better options in terms of cost per unit-speedup.

D. Discussion

The experiments demonstrate some interesting tradeoffs between cost and performance, the requirement for persistence, and the average unit-data size. We summarize these options below, given parameters d = average unit-data size, T = total cache size, and R cache requests per month.

For smaller data sizes, i.e., $d \leq 5$ MB, and small total cache sizes $T < 2$ GB, we posit that because of its affordability, `S3` offers the best cost tradeoff when R is small, even for supporting volatile caches. `m1.small.mem` and `m1.small.disk` also offer very good cost-performance regardless of the request rate, R. This is due to the fact that the entire cache can be stored in memory, together with the low cost of `m1.small` allocation. Even if the total cache size, T, is much larger than 2 GB, then depending on costs, it may still even make sense to allocate multiple small instances and still store everything in memory, rather than using one small instance's disk – we showed that, if request rate R is high, and the unit-size, d, is small, the speedup for `m1.small.disk` is eventually capped two orders of magnitude below the memory-bound option. If $d \geq 50$ MB, we believe it would be wise to consider `m1.xlarge`. While it could still make sense to use a single small instance's disk if R is low, we observed that performance is lost quickly as R increases, due to `m1.small`'s lower-end I/O.

If data persistence is necessary, `S3` is by far the most cost-effective option in most cases. However, it also comes at the cost of lower throughput, and thus `S3` would be viable for systems with less expectations for high amounts of requests. The cost analysis also showed that storage costs are almost negligible for `S3` and `EBS` if request rates are high. If performance is an issue, it would be prudent to consider `m1.small-ebs` and `m1.xlarge-ebs` for smaller and larger unit-data sizes respectively, regardless of the total cache size. Of course, if cost is not an a pressing issue, `m1.xlarge` with or without EBS persistence should be used achieve the highest performance.

IV. Related Works

Since its emergence, there has been growing amounts of interest in testing the feasibility of performing data- and compute-intensive analysis in the Cloud. For instance, a plethora of efforts have been focused around the popular MapReduce paradigm [8], which is being supported in various Cloud frameworks, including Google AppEngine [10], AWS Elastic MapReduce [5], among others. One early experience with scientific workflows in the Cloud is discussed in [11]. They showed that the running their scientific workflow over the Cloud was comparable to performance in a cluster, albeit that certain configuration overheads do exist in the Cloud. This specific application is among several others [17], [15], [13] that have been mapped onto the Cloud.

While Cloud-based data-intensive applications continues to grow, the cost of computing is of chief importance [3]. Several efforts have been made to assess costs for various large-scaled projects. Kondo, *et al.* compared cost-effectiveness of AWS against volunteer grids with the [12]. Deelman, *et al.* reported the cost of utilizing Cloud resources to support a representative workflow application, Montage [9]. They reported that CPU allocation costs will typically account for most of the cost while storage costs are amortized. Because of this, they found that it would be extremely cost effective to cache intermediate results in Cloud storage. Palankar *et al.* conducted an in-depth study on using S3 for supporting large-scale computing [14]. Our results on S3 echoed their findings, and we agree that S3 can be considered when average data size is larger.

In terms of utilizing the Cloud for caching/reusing data, Yuan, *et al.* proposed a strategy [19] for caching for large-scale scientific workflows. Their system decides to store or evict intermediate data produced at various stages of a workflow, based on cost tradeoffs between storage and recomputation and reuse likelihood. Their analysis ignores network transfer costs, which we showed to be a dominant factor. A tangential study by Adams *et al.* discussed the potentials of trading storage for computation [1]. Weissman and Ramakrishnan discussed deploying Cloud proxies [18] for accelerating web services. Servers close to the computation are used to store intermediate data. Our work differs from the above in that, we are evaluating various cache storage options in the Cloud and projecting the cost, performance, and persistence tradeoffs of each.

V. Concluding Remarks

As large-scale applications are becoming increasingly data-intensive, caching and data storage strategies can help increase their performance. Using the Cloud, which has become increasingly popular, can offer an on-demand framework for supporting such data caches. However, a major issue is the cost that comes coupled of utilizing several mixed options to support a cache. Depending on application parameters and needs, we have shown that certain scenarios call for different Cloud resources.

We hope to use this study to initiate the development of finer-grained cost models and automatic configuration of such caches given user parameters. We will also develop systematic approaches, including hybrid cache configurations to optimize cost-performance tradeoffs.

References

[1] I. F. Adams, D. D. Long, E. L. Miller, S. Pasupathy, and M. W. Storer. Maximizing efficiency by trading storage for computation. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[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] Amazon web services, http://aws.amazon.com.

[5] Amazon elastic mapreduce, http://aws.amazon.com/elasticmapreduce/.

[6] D. Chiu and G. Agrawal. Hierarchical caches for grid workflows. In *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 2009.

[7] D. Chiu, A. Shetty, and G. Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Supercomputing 2010 (SC'10)*, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

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

[11] 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, Washington, DC, USA, 2008. IEEE Computer Society.

[12] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[13] J. Li, *et al.* escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[14] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[15] 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.

[16] 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.

[17] 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.

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

[19] D. Yuan, Y. Yang, X. Liu, and J. Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.