Supporting Placement and Data Consistency Strategies using Hybrid Clouds

Chris Bunch Navraj Chohan Chandra Krintz
Department of Computer Science
University of California, Santa Barbara

Abstract—**In this paper, we investigate cloud platform support that provides distributed applications with automatic service placement across different cloud computing systems (hybrid clouds), and that enables application developers to investigate the impact of using different cloud-based data consistency models with their applications. By pursuing its implementation at the cloud runtime layer (platform), we are able to provide hybrid cloud support without requiring application modification or significant developer expertise.**

We investigate the efficacy of such portabililty for different application domains (web services and computationally-intensive HPC applications, via Monte Carlo simulations). We examine different hybrid cloud placement strategies based on cost, performance, and common use cases. We evaluate the performance of such placement strategies as well as different data consistency policies.

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION

Distributed computing has long offered great promise to programmers interested in developing applications that serve large volumes of web traffic or that operate on vast data sets. However, accessing large numbers of machines has traditionally been expensive enough to limit their use severely by programmers at large. Cloud computing, a service-oriented approach to distributed systems, has done much to rectify this problem. Vendors of cloud computing services provide access to a well-defined infrastructure, platforms, or software and charge for its use using pre-determined service-level agreements (SLAs), at a fraction of the cost of owning and maintaining a cluster.

An SLA guarantees a specified level of service quality and/or availability (e.g., 10Gb/s maximum IO performance or 99.9% uptime). If the guarantees are not met, the user can request a partial refund. In practice, such refunds are of little use: the cost of the service that has failed is often insignificant compared to the costs suffered by the business operating them (e.g., lost business for web vendors). Moreover, the onus of reporting guarantee violation is on the customer not the provider.

Although infrequent, all of the major cloud computing vendors have suffered from service outages lasting from several hours to several days. Amazon Web Services (AWS), an infrastructure provider, offers access to virtual machines that users can customize for their own applications. On April 21, 2011, machines hosted in their U.S. East Coast datacenters failed, causing an outage of multiple hours for most users and multiple days for others [1]. Businesses hosted solely on AWS's East Coast datacenters were unavailable for the duration of the outage. A noteworthy business that experienced an outage due to this was Heroku, a cloud platform provider that offers scalable hosting for Ruby on Rails applications. As Heroku was hosted largely within the AWS East Coast, Amazon's failure caused their applications to fail, and customers using Heroku for hosting suffered from 16 to 60 hours of operational downtime [2]. Google also provides a platform offering, known as App Engine, and as it is hosted entirely on machines that Google owns. On February 24, 2010, App Engine suffered from a Datastore outage that prevented users from writing data to the database that App Engine utilizes for the duration of a day [3]. Other failures across these providers have been detailed in [4] [5][6].

To enable applications to tolerate outages such as these, cloud vendors increasingly offer services that span multiple data centers. For example, Amazon Elastic Compute Cloud (EC2) users can host their applications in multiple datacenters (e.g., the U.S. West Coast and U.S. East Coast) so that a single datacenter's failure does not cause the failure of all machines hosting the application. Doing so, however, is a manual exercise and requires significant user expertise. The Google App Engine cloud platform automates such use for hosted applications, which can be written in Python, Java, and Go. Most services that App Engine hosts only require the use of a single datacenter, but they do offer a structured data storage service known as the High Replication Datastore that spans multiple datacenters. These services are only offered on-premise, though, and are not available to run within a locally maintained datacenter.

Although these clouds have largely been used for web services to date, they are seeing additional use for scientific computing. One particular problem domain that has found use is the Monte Carlo simulation, by which a process is simulated via a large number of probalistic, independent computations. As the computations do not rely on one another, they follow an embarrassingly parallel programming model and can be executed in parallel. The technique is general enough that it has seen widespread use: some uses within the Aerospace community include target tracking of aircraft [7], flight safety risk assessments [8], and aerospace vehicle structure simulations [9]. The popular MapReduce [10] programming

978-1-4577-0557-1/12/$26.00 ©2012 IEEE.

paradigm enables support for distributed Monte Carlo simulations, and thus can be executed over Amazon and Google App Engine through the Elastic MapReduce and MapReduce API offerings, respectively.

Developers can take steps toward fault tolerance and cross-cloud portability (the use of different cloud fabrics or lower cost services by the same application) by designing their applications for execution across different cloud systems, a hybrid cloud deployment. This is possible in theory since most cloud vendors today offer a similar subset of services (e.g. un/structured data management, instance control, load balancing, elasticity, programming models ala map-reduce, background task execution, etc.). Unfortunately, cloud vendors are disincentivized to interoperate with each other since doing so will enable users to move between vendors more easily, thus increasing competition. Instead, cloud vendors "lock-in" users through the use of different service APIs, control tools (APIs), and SLAs, which make practical hybrid cloud use very challenging and requiring significant application modification and complexity.

In this work, we investigate enabling hybrid cloud use via cloud platform layer. To do so we extend the AppScale [11], [12], [13] cloud platform – a free and open source cloud runtime system that executes over different cloud fabrics. Applications that use the AppScale APIs avoid lock-in since they can execute on any cloud fabric over which AppScale runs, i.e. the AppScale platform provides a portability layer for applications. We use this platform to investigate the potential of hybrid cloud deployments. In addition, we extend the system to enable developers to control and to investigate the characteristics of two key cloud services: consistency of structured data and service/component placement.

Cloud providers today offer access to structured data via distributed and highly available key-value datastore technologies. Updates to data using these technologies can be strongly consistent (ala Google BigTable [14]), i.e., any read that follows a write is guaranteed to see the updated value. When there are multiple copies of the data that must be kept consistent, this constraint can introduce overhead and limit the degree to which such data access scales. As an alternative, the datastore can relax this constraint so that updates are "eventually" propagated through the system (ala Amazon Dynamo [15]). Therefore, a read by some other user may see an older, stale value instead of the most recent update. Datastores implement eventual consistency by reducing the number of replicas required to read or write a piece of data. Although some applications require strong consistency guarantees, many do not and thus can benefit from the additional scale that relaxed consistency offers. However, since eventual consistency is a relatively new option, many developers are unsure of how their applications might be affected by its use. Our first AppScale extension enables developers to experiment with different consistency levels offered by different cloud datastore technologies.

Service placement is a mechanism for executing components that an application uses on specific physical or virtual machines (VMs). Intelligent placement of components can improve both communication and computation performance, e.g. heavily communicating components can be placed "near" each other to reduce latencies. Component placement within VMs relies on process-level isolation in contrast to the full isolation that VM isolation provides. Like consistency, different applications can exploit and benefit from different placement configurations and trade offs in different ways. However, to date there is no automated approach to intelligent placement across cloud systems. Thus, providing such placement for an application must be done manually by each individual applications developer, requiring significant learning curves, expertise, and deployment/configuration complexity. Our second AppScale extension enables developers to experiment with different placement options within the same [16] and across different cloud fabrics (this work), easily and without application modification.

We implement hybrid cloud support and facilitate experimentation with data consistency and service placement for both web service and high-performance computing (HPC) application domains. We examine different hybrid cloud placement strategies based on cost, performance, and common use cases. We evaluate the performance of such placement strategies as well as different data consistency policies using real applications and disparate cloud fabrics. We find that our extensions significantly simplify hybrid cloud application deployment and lay the groundwork for cross-cloud application portability, concurrent use, and improved fault tolerance. In summary, with this paper we contribute:

- An automated web hosting platform that operates over hybrid clouds without requiring application designers to be cognizant of the number of infrastructures or the characteristics of the infrastructures it runs over.
- Hybrid cloud support for strong and eventual consistency for applications hosted within AppScale.
- Service placement techniques for hybrid cloud deployments, supporting both web applications and HPC applications within AppScale.
- An evaluation of the consistency support and placement support over hybrid cloud deployments enabled via AppScale, utilizing a web application benchmark and a set of Monte Carlo simulations similar in nature to those encountered by the Aerospace community.

In the sections that follow, we present our AppScale extensions for hybrid cloud data consistency configuration (Section 2), and service placement (Section 3). We then present our empirical use cases, evaluation, and analysis of our extensions (Section 4). In Sections 5 and 6, we overview related work and conclude.

2. Hybrid Cloud Consistency Support

Traditionally, programmers have designed systems that operate within a single datacenter and operate on data with strong consistency guarantees. Given recent advances in cloud computing technologies both of these assumptions can be relaxed to offer greater scalability and fault tolerance. To help programmers understand, evaluate, and exploit these advances, we investigate cloud platform support that allows developers to experiment with different consistency models and multi-cloud (hybrid) distributed application placement – without requiring them to modify their applications. We first investigate such support for differing data consistency models and then consider hybrid cloud service placement.

Data Consistency Models

Non-relational datastores, commonly referred to as NoSQL datastores, have emerged in the last decade to offer greater scalability at the cost of a smaller feature set than those offered by relational databases such as MySQL. One notable feature is eventual consistency: instead of a read or write operation requiring agreement between all holders of the data (known as replicas), only a subset of the replicas need be contacted. This can result in a system that is eventually consistent: data recently written may not be available to all nodes in the system, and readers may receive a stale piece of data. Gossip protocols [17] are used to update the data on the other replicas after the initial write has been performed, but the application must take the possibility into account of receiving stale data. This is not always acceptable: some applications require strong consistency at all times.

Two NoSQL datastores have emerged with support for variable consistency: Cassandra [18] and Voldemort [19]. These systems are variably consistent: specifically, the user has control over the type of consistency used in the system. The system can be either strongly consistent or eventually consistent, and both datastores provide a number of relevant configuration settings. With the Cassandra datastore, read and write consistency is specified by the application itself, indicating that a particular request must reach consensus by agreement between all replicas, a majority of replicas (a quorum), or only a single replica. A read request returns a given piece of data but no indication of whether or not it is stale (as the system may not know this information). In contrast, the Voldemort datastore requires the database administrator to create a database with a statically assigned read and write factor (that is, the number of nodes required for consensus in a read or write operation), and if multiple versions exist for a piece of data, they are given to the user with timestamp information.

These systems provide users with mechanisms for both eventual consistency and improved performance on strongly consistent applications in certain scenarios. For example, if it is known that the database will be used for predominantly read operations, read operations can be set to succeed via contacting only a single replica, and forcing write operations to contact all replicas. This speeds up read operations at the cost of write operations, and maintains a strongly consistent system. If the distribution of reads to writes is fairly equal, a quorum of replicas can be used for both operations to ensure similar performance on these operations. If the application can tolerate eventual consistency, then it can run with only a single replica needed for read and write operations.

Variable Consistency via AppScale

AppScale currently provides support for the Cassandra and Voldemort datastores, but by default configures them to use strong consistency. This is done because a given user's application may require strong consistency and may break or act incorrectly in eventually consistent modes. In this work, we extend AppScale's command-line tools, used by system administrators to start and stop AppScale deployments, to also allow users to specify the read and write policy that applications should use.

For the Cassandra datastore, users can specify that any of Cassandra's acceptable read or write factors (one machine, a quorum of replicas, or all replicas for consensus) be used. In Voldemort, users specify the number of replicas required for a read or write operation. Users can also specify the number of replicas required for a given piece of data. This allows users to specify that only a single replica is required and to keep fast, strongly consistent access to their data, at the cost of durability. This also allows users to quantify the cost of varying the read, write, and replication factors within a single cloud deployment or hybrid cloud deployments. It also enables users to automatically deploy applications across multiple clouds and not need to code the consistency model or replication model into their applications.

3. Hybrid Cloud Service Placement

Service placement traditionally refers to the ability to place essential services on the same machine or different machines. In the web services domain, this typically entails the placement of load balancers, web servers, and database nodes, and in the HPC domain, this also entails the placement of compute nodes. We extend this to include the placement of cloud services. In particular, we provide support for users to specify such placement programmatically and have the underlying platform or infrastructure configure the services automatically as part of the application deployment process.

Service Placement in Cloud Providers

Before looking at how we add hybrid cloud placement support to AppScale, we first examine existing solutions offered by cloud computing vendors. Amazon offers its Elastic Load Balancing [20] and SimpleDB [21] services to provide optimized load balancing and database services, respectively. These services are fully managed by Amazon. Both can operate within multiple datacenters within a single region (e.g., the U.S. East Coast), but users must write their own service to replicate data across regions if needed.

Heroku and Google App Engine, operating at the platform layer, allow users to specify the number of web servers that should be used for the user's application. The former exposes these controls via a command-line interface while the latter does so via a web interface. Both do so to give users the ability to limit the monetary costs they incur by running their application with a bounded number of resources or to scale up if more machines are required.

Extending AppScale with Hybrid Cloud Service Placement

AppScale goes one step further than Heroku and Google App Engine as a research platform and allows arbitrary services to be placed on the same machine (colocated) if requested. It also allows for the placement of all services in the system, including `memcache` for caching support, database nodes, and hot spares to be used in the platform as needed.

Before this work, however, placement support was largely limited to a single cloud. AppScale does allow for dynamic scaling of web servers and compute nodes via the Neptune domain-specific language, and this support does extend to hybrid cloud deployments. However, it did not allow the user to statically specify the placement for hybrid cloud deployments.

In this work, we extend the command-line tools that AppScale provides for cloud administrators (known as the AppScale tools) to allow them to specify hybrid cloud placement in a manner identical to that of single cloud placement. As an example, consider the scenario in which a single cloud is used to run AppScale with two web servers and two database nodes, shown in Figure 1. Configuration files are specified in YAML [22], a markup language similar in use to XML. The configuration file needed for this placement strategy would be:

```
---
:master: node-1
:appengine:
- node-1
- node-2
```

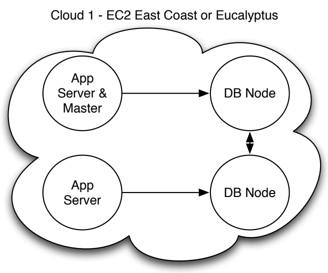

Figure 1. A single cloud deployment that runs application servers and database nodes in Amazon EC2 or Eucalyptus.

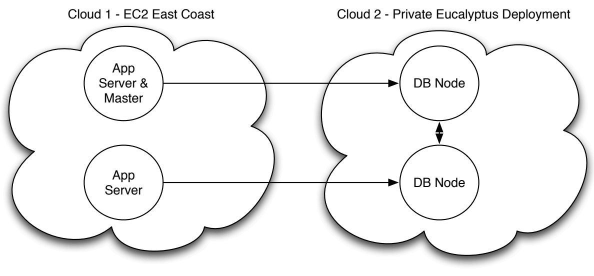

Figure 2. A hybrid cloud deployment that runs application servers in Amazon EC2's East Coast region and database nodes in a privately maintained Eucalyptus cloud.

```
:database:
– node–3
– node–4
```

This deployment also includes the use of a single node (named `master` in these examples) that handles fault-tolerance and database transaction semantics. The analogous configuration file for a hybrid cloud deployment utilizing two clouds (visually depicted in Figure 2) would be:

```
–––
:master: cloud1–1
:appengine:
– cloud1–1
– cloud1–2
:database:
– cloud2–1
– cloud2–2
```

This configuration file in particular also specifies that all of the application servers should be placed in the first cloud and all of the database nodes should be placed in the second cloud. One foreseeable use case for this type of deployment would be for scenarios where the data computed by the application server or compute node is sensitive data that cannot be stored on an externally managed cloud like Amazon EC2. Here, the database nodes all run in a separate cloud, so users could set up a locally maintained Eucalyptus cloud [23] that is managed by trusted users.

If the user wanted to place one application server and one database node in each cloud, the configuration file would be slightly different (visually depicted in Figure 3):

```
–––
:master: cloud1–1
:appengine:
– cloud1–1
– cloud2–1
:database:
– cloud1–2
– cloud2–2
```

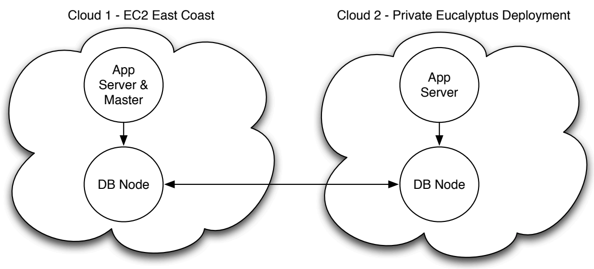

Figure 3. A hybrid cloud deployment that runs one application server and one database node in each of two clouds, Amazon EC2 and Eucalyptus.

Once the user specifies their credentials for each cloud (via environment variables), the AppScale tools spawn the node marked `master` in whichever cloud has been specified. The tools then give this node the credentials and the placement file and instruct it to spawn the other nodes as needed. This workflow is identical to the single cloud scenario, with the exception that now more than a single cloud is supported. The user's web applications or HPC code does not need to be aware of how many clouds are present: it is abstracted away and handled by AppScale and Neptune.

4. Evaluation

We next evaluate our extensions to AppScale and use them to investigate the performance impact of utilizing eventual consistency and placement support in hybrid cloud environments. We first present our experimental methodology and then discuss our results.

Experimental Methodology

To evaluate the costs associated with eventual consistency and placement support in hybrid cloud environments, we benchmark their use with web service and HPC applications. Across all of our experiments, we run AppScale over a hybrid cloud deployment utilizing a locally maintained Eucalyptus cloud (located on the U.S. West Coast) and a remotely maintained Amazon EC2 cloud (located on the U.S. East Coast). The Eucalyptus cloud runs Eucalyptus 2.0.3 with one physical machine dedicated to use as a Cloud Controller, Cluster Controller, and Storage Controller, while all other machines involved act solely as Node Controllers. We utilize AppScale 1.5.1, modified to employ our hybrid cloud support for user-specified eventual consistency and placement support. Neptune 0.1.1 and Cassandra 0.7.6-2 are also utilized throughout these experiments. All values reported here represent the average of five runs.

The web service application we use is Active Cloud DB [24]. Active Cloud DB is a Google App Engine application written in Python that provides a RESTful interface to any database with support for the Google Datastore API. It also employs the Google Memcache API to provide a write-through cache to speed up database puts, gets, and deletes, and a generational cache for query operations. For these experiments, we have disabled its use of the Memcache API to better understand the operation of the underlying database. In all web service experiments, we dispatch 10,000 web operations, each being either a read or write operation, to Active Cloud DB. Reads and writes are sent in serial and at a 20:3 ratio, a rate that is consistent with the web application access rate measured in [25]. For writes, we also vary the size of the data written, choosing randomly to write either 10, 100, or 500 characters for each entry (500 characters being the limit for the App Engine string database type used by Active Cloud DB).

We fix the placement strategy used to place three database nodes in Eucalyptus and an application server in Amazon EC2, use 2-times replication, and vary the data consistency policy used between four policies: an eventually consistent policy (read=one, write=one), a strongly consistent policy favoring reads (read=one, write=all), a strongly consistent policy favoring writes (read=all, write=one), and a strongly consistent policy favoring neither reads nor writes (read=quorum, write=quorum). We refer to these in the results as `Eventually Consistent`, `Reads Favored`, `Writes Favored`, and `Read/Write Neutral`, respectively. As data is replicated twice over three database nodes, the quorum case for reads and writes is equivalent to the read-all, write-all scenario.

Our HPC application is a specialized type of Monte Carlo simulation, known as a Stochastic Simulation Algorithm (SSA) [26]. This algorithm, also referred to as a kinetic Monte Carlo algorithm, is used to simulate the conditions found in biochemical systems, and follows a similar execution strategy to Monte Carlo simulations used by the Aerospace community. The particular SSA we utilize in this work is known as the Diffusive Finite State Projection Algorithm [27], and like other Monte Carlo algorithms, is structured in an embarrassingly parallel manner: it runs a number of Monte Carlo simulations and takes the average of the results seen. Previous works [16] have executed 10,000 simulations to gain statistical accuracy: here we run 1,000 simulations, as we are more interested in potential performance gains from altering service placement or the data consistency policy than focusing on the computation itself. We measure both the amount of time spent executing simulations as well as the amount of time spent storing the results of these simulations. We currently store each result individually after it has been calculated (providing a total of 1,000 smaller storage operations) as opposed to storing the results after all computations have been completed (providing a total of one large storage operation). This is done for fault-tolerance, so that the failure of a single node does not entail the loss of all the simulations it has computed.

We run these experiments first utilizing strong consistency favoring writes (read=all, write=one), as this workload is write-heavy. We then repeat the experiment under an eventually consistent model (read=one, write=one). We run each of these experiments under two placement strategies. The first (hereafter referred to as Config 1) places three database nodes in Eucalyptus and two compute nodes in Amazon EC2 (similar to that of Figure 2). The second (hereafter referred to as Config 2) places two database nodes in Eucalyptus, one in Amazon EC2, and places one compute node in Eucalyptus, and another in Amazon EC2 (similar to that of Figure 3).

Results

We begin by examining the results of Active Cloud DB between the four data consistency policies described earlier. Table 1 shows the average latency and throughput for 10,000 randomly dispatched operations. Read/write neutral performs the worst of the policies examined, while the other three policies perform similarly to one another. This concurrence is due to the fact that these policies require communication with only a single node for a read and/or write, and do not need to dispatch requests to all nodes and wait for a majority to respond successfully.

	Response Time (sec/op)	Throughput (op/sec)
Eventually Consistent	0.672 ± 0.002	1.486 ± 0.004
Reads Favored	0.676 ± 0.001	1.477 ± 0.002
Writes Favored	0.670 ± 0.001	1.492 ± 0.002
Read/Write Neutral	0.699 ± 0.001	1.430 ± 0.002

Table 1. Average response time and throughput for the Active Cloud DB benchmark with different data consistency settings under low load.

	Response Time (sec/op)	Throughput (op/sec)
Eventually Consistent	5.916 ± 0.009	0.169 ± 0.0002
Read/Write Neutral	6.030 ± 0.003	0.165 ± 0.0001

Table 2. Average response time and throughput for the Active Cloud DB benchmark with different data consistency settings under high load.

All 10,000 web requests dispatched in each experiment caused the application to return an HTTP 200 response, which indicates that the system was able to serve all web requests without error. This is to be expected: a single accessing thread does not provide enough traffic on its own to overburden the multiple load balancers, application servers, and database nodes that are host this application. However, it is enough traffic to demonstrate a difference in performance between the read/write neutral policy and the other policies. It is notable that the eventually consistent policy does not perform significantly faster than the read or write-favoring policies. This is expected to be because of the relatively low amount of load we are putting on the system, and we expect that sending requests to the system in parallel would demonstrate a larger difference between these policies.

We continue our web experiment by increasing the amount of load on the system from a single machine with a single accessing thread to three machines with three accessing threads each, for a total of nine accessors. Each accessor performs 10,000 web requests, the results of which are shown in Table 2. As opposed to the previous experiment, we limit ourselves to the eventually consistent policy and the strongly consistent read-write-neutral policy. As was the case previously, all requests still return HTTP 200 codes, indicating that all requests were successful, but we see a near-linear slowdown with respect to response time and throughput. This provides enough load to where we can see a more pronounced difference between the eventually consistent policy and the read-write-neutral policy, in favor of eventual consistency.

We next examine the results of our SSA implementation across the two placement strategies that we described earlier. Here we compare strong consistency against eventual consistency. For the DFSP code (an SSA implementation), we measure both the total computation time as well as the

	Strongly Consistent (secs)	Eventually Consistent (secs)
Config 1	4747.69 ± 348.18	5086.36 ± 223.21
Config 2	4655.25 ± 265.46	4680.76 ± 280.70

Table 3. Total computation times for the DFSP SSA implementation with different placement strategies and data consistency settings.

	Strongly Consistent (secs)	Eventually Consistent (secs)
Config 1	1458.38 ± 55.48	1781.38 ± 298.66
Config 2	1334.73 ± 20.76	1691.03 ± 497.40

Table 4. Total storage times for the DFSP SSA implementation with different placement strategies and data consistency settings.

total amount of time spent storing data to Cassandra, referred to as the total storage time. The total computation time for these experiments is shown in Table 3, while the total storage time is shown in Table 4. As expected, the total computation time is similar regardless of which placement strategy or data consistency setting is used (as it is independent of these factors). However, the variance in the results has significantly increased when we use a hybrid placement that includes Amazon EC2.

We believe that this is due to the performance fluctuations of Amazon EC2 itself as opposed to our extensions and system. We have observed that we occasionally receive machines that performed much slower than average. When we run this experiment on machines that were either consistently fast or slow produced results with an order of magnitude less standard deviation (e.g., 20 seconds versus 200 seconds). As this required us to acquire new machines for our experiment for each of the five runs the data is averaged over, using a single set of machines for all of the five runs is equivalent to acquiring five equally-powered machines, and thus also exhibits low standard deviation.

This behavior is not identical with respect to the total storage time: here, we only saw large standard deviations in the cases where eventual consistency was used. This is likely due to our execution/use of Cassandra's gossip protocol over a hybrid cloud environment, as we were not utilizing Cassandra's cross-datacenter capabilities. We are looking into enabling automated support for this and validating this hypothesis.

Regardless, we do see that using eventual consistency does not provide a significant improvement in execution or storage time, and that neither placement strategy significantly improves the storage time. This is for the same reason as in the web experiment: the DFSP code does not produce enough output at a fast enough rate to burden the database to the point where eventual consistency or a different placement strategy would be effective.

5. Related Work

Other works exist that provide support for hybrid cloud deployments. As Google App Engine automatically and transparently hosts web applications across multiple datacenters, it is the closest to our work in spirit. Its High Replication Datastore (HRD) [28] is perhaps the largest-scale automated hybrid cloud deployment, leveraging multiple datacenters to provide data redundancy and mask the impact of a failed or performance-degraded datacenter. HRD also prevents programmers from having to code their applications in a manner that forces them to be aware of how many resources they run over and where the resources are.

Our extensions to AppScale are done with the same motivation: to provide those looking to do research on cloud platforms with a way to run applications over hybrid cloud resources. In particular, we emphasize that the application must not need to be aware of where it is located and how many resources it utilizes. In contrast to App Engine, we provide users with a way to explicitly dictate where all of the critical services run, allowing them to experiment with different configurations with respect to monetary cost, performance, or their own specific use cases. App Engine also primarily targets the web service domain, and while AppScale does as well, it also provides support for HPC frameworks.

App Engine also does allow applications to set their own read and write policies, but overrides this behavior within transactions (transactions always are strongly consistent). This forces the application designer to specify the consistency settings for each read and write (if different from the default setting). Our extensions take a different approach, and allow a given read and write policy to be set for applications transparently so that they do not have to change their code to make use of it.

6. Conclusion

We contribute a set of extensions to the AppScale cloud platform that facilitate portable use and automatic configuration of hybrid cloud environments. We extend this portability to the data management level to offer a choice of eventual and strong consistency to application developers. Specifically, we enable users to set read and write consistency settings and automatically apply them to their application. This is done below the application's level of abstraction, and does not require the application's code to change to accomodate it. We also allow cloud administrators to specify placement strategies to be used within hybrid cloud environments, and automatically deploy their services accordingly. This enables users to quantify the effects of different levels of consistency and different placement strategies within hybrid cloud environments.

These contributions aid both web application developers as well as the Aerospace community, who utilize Monte Carlo simulations like those shown here in their own work. This gives users an open platform with which they can experiment with via different data consistency models and hybrid cloud environments. Our modifications to AppScale have been committed back to the AppScale project and can be found at `http://appscale.cs.ucsb.edu`.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was funded in part by Google, IBM, and NSF grants CNS-CAREER-0546737 and CNS-0905237.

REFERENCES

[1] "Final Thoughts on the Five-Day AWS Outage," http://www.eweek.com/c/a/Cloud-Computing/Final-Thoughts-on-the-FiveDay-AWS-Outage-236462/.

[2] "Heroku: Widespread Application Outage," https://status.heroku.com/incident/151.

[3] "Unscheduled App Engine Outage - February 24th, 2010," http://groups.google.com/group/google-appengine-downtime-notify/browse_thread/thread/b4ed491a8b9ccce2.

[4] "Amazon Web Services Goes Down, Takes Many Startup Sites With It," http://techcrunch.com/2008/02/15/amazon-web-services-goes-down-takes-many-startup-sites-with-it/.

[5] "Detailing Cloud Downtime, AWS Releases Report," http://www.thehostingnews.com/detailing-cloud-downtime-aws-releases-report-19876.html.

[6] "Postmortem for August 18, 2011 outage," http://groups.google.com/group/google-appengine-downtime-notify/browse_thread/thread/4f0be3699386a305.

[7] J. Vermaak, S. Godsill, and P. Perez, "Monte Carlo Filtering for Multi-Target Tracking and Data Association," in *IEEE Transactions on Aerospace and Electronic Systems*, 2005.

[8] P. Wang and T. Jin, "Flight safety risk assessment using Monte Carlo simulation - A real case study," in *International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*, 2011.

[9] R. Vaicaitis, "Monte Carlo Simulation for Appliction to Aerospace Vehicle Structures," 1990.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)*, pp. 137–150, 2004.

[11] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "AppScale: Scalable and Open AppEngine Application Development and Deployment," in *ICST International Conference on Cloud Computing*, Oct. 2009.

[12] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura, "An Evaluation of Distributed Datastores Using the AppScale Cloud Platform," in *IEEE International Conference on Cloud Computing*, Jul. 2010.

[13] N. Chohan, C. Bunch, C. Krintz, and Y. Nomura, "Database-Agnostic Transaction Support for Cloud Infrastructures," in *IEEE International Conference on Cloud Computing*, Jul. 2011.

[14] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Symposium on Operating System Design and Implementation*, 2006.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Symposium on Operating System Principles*, 2007.

[16] C. Bunch, N. Chohan, C. Krintz, and K. Shams, "Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms," in *ACM 2nd Workshop on Scientific Cloud Computing*, 2011.

[17] R. Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient Reconciliation and Flow Control for Anti-Entropy Protocols," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008.

[18] "Cassandra," http://incubator.apache.org/cassandra/.

[19] "Voldemort," http://project-voldemort.com/.

[20] "Amazon Elastic Load Balancing," http://aws.amazon.com/elasticloadbalancing/.

[21] "Amazon SimpleDB," http://aws.amazon.com/simpledb/.

[22] "YAML," http://yaml.org/.

[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," in *IEEE International Symposium on Cluster Computing and the Grid*, 2009, http://open.eucalyptus.com/documents/ccgrid2009.pdf.

[24] C. Bunch, J. Kupferman, and C. Krintz, "Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores," in *ICST International Conference on Cloud Computing*, 2010.

[25] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *5th Biennial Conference for Innovative Data Systems Research*, 2011.

[26] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *J. Phys. Chem.*, vol. 81, no. 25, pp. 2340–2361, 1977.

[27] B. Drawert, M. J. Lawson, L. Petzold, and M. Khammash, "The diffusive finite state projection algorithm for effficient simulation of the stochastic reaction-diffusion master equation," *J. Phys. Chem.*, vol. 132, no. 7, 2010.

[28] "Announcing the High Replication Datastore for App Engine," http://googleappengine.blogspot.com/2011/01/announcing-high-replication-datastore.html.

BIOGRAPHY

Chris Bunch is a fifth-year Ph.D. student at the University of California, Santa Barbara. His research interests include language support for emerging virtualization-based systems (e.g., cloud computing), with an emphasis on accessibility and ease-of-use for programmers with varying skill levels and backgrounds. As such, he is interested in how cloud computing platforms intersects with other domains such as scientific computing and high-performance computing. His other interests include contributing software artifacts that he develops to the open-source and research communities. Recent projects he has been involved with are AppScale, an open source implementation of the Google App Engine APIs, and Neptune, a domain specific language for configuring and deploying HPC applications.

Navraj Chohan Navraj Chohan is currently a Ph.D. candidate at the University of California, Santa Barbara, where he has received his B.S. and M.S. in computer engineering. He has worked at Applied Signal Technologies as a software engineer, and at T.J. Watson IBM research, as well as Lawrence Livermore National Lab as a research intern. His interests are in distributed systems, ranging from sensor networks to cloud computing. In recent times he has worked on AppScale, a private cloud platform, compatible with the Google App Engine APIs.

Chandra Krintz is a Professor of Computer Science at the University of California, Santa Barbara (UCSB). She joined the UCSB faculty in 2001 after receiving her M.S. and Ph.D. degrees in Computer Science from the University of California, San Diego (UCSD). Chandra's research interests include automatic and adaptive compiler, programming language, virtual runtime, and operating system techniques that improve performance (for high-end systems) and that increase battery life (for mobile, resource-constrained devices). Her recent work focuses on programming language and runtime support for cloud computing. Her group has recently developed and released AppScale – an open-source platform-as-a-service cloud computing system that implements the Google App Engine (GAE) APIs that facilitates next-generation cloud computing research. Chandra has supervised and mentored over 50 students, has published her work in a wide range of ACM venues including CGO, ECOOP, PACT, PLDI, OOPSLA, ASPLOS, and others, and leads several educational and outreach programs that introduce computer science to young people, particularly those from underrepresented groups. Chandra's efforts have been recognized with a NSF CAREER award, the CRA-W Anita Borg Early Career Award (BECA), and the UCSB Academic Senate Distinguished Teaching Award. Chandra is also an ACM and IEEE Senior Member.