MEDEA: A Pluggable Middleware System for Interoperable Program Execution Across Cloud Fabrics

Chris Bunch Brian Drawert Navraj Chohan Andres Riofrio
Chandra Krintz Linda Petzold
Computer Science Department
University of California, Santa Barbara
UCSB Technical Report #2012-11, October 2012

1. ABSTRACT

In this paper we present MEDEA, a software architecture that abstracts away the details of cloud infrastructures to make it easy to deploy applications over disparate fabrics without requiring application modification or expertise with the underlying technologies. MEDEA consists of scripting language support with which developers describe the inputs, outputs, and location of their code, as well as their cloud credentials. This support then employs a remote web service that packages applications so that they can be executed as background tasks over different cloud infrastructures. This execution engine integrates pluggable components for task queues, task workers, and storage implementations. We then plug into this framework extant cloud infrastructure implementations from Microsoft Azure, Amazon Web Services, Eucalyptus, and Google App Engine. MEDEA helps developers avoid lock-in, compare and contrast different cloud offerings (their restrictions, costs, performance), and evaluate hybrid cloud deployments (furthering cloud interoperability).

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Software Engineering - Language Classifications (Extensible Languages); C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks - Distributed Systems (Distributed Applications)

General Terms

Design, Languages, Performance

Keywords

Cloud Platform, PaaS, Open-Source, Distributed Systems

2. INTRODUCTION

Cloud infrastructures provide intuitive, utility-style access to pools of on-premise or publicly available resources (compute, storage, networking, and software services). Although accessing cloud resources is cheap and readily available, doing so from cloud infrastructure providers requires significant expertise, experience, and time for customization, configuration, deployment, and management of virtual machines (VMs). Despite the emergence of technologies that aid users in these endeavors, significant challenges remain with making effective use of federated cloud systems.

One key challenge is the multitude of cloud offerings. With each comes different application programming interfaces (APIs), toolkits, scale, service level guarantees, performance levels, pricing models, rules, and restrictions. This diversity of offerings makes it challenging for new and expert software developers to determine which set of services is best for their apps, for some definition of "best" (e.g. price, performance, scale, configurability, familiarity, ease of use). In addition, once a developer invests the time to learn a particular system and codes their applications to it, they become "locked in" to the technology since moving to a competitive offering requires rewriting their applications and learning the intricacies of a different technology (however similar in overall function). The lack of portability across clouds and cloud services consume developer time and focus that could instead be used for the core innovation and science in the applications themselves.

The goal of our work is to design and develop an open source software framework that reduces the barrier to entry on the use of cloud infrastructures and that facilitates their use and interoperation for a wide range of domains and application domains. As a first step in this direction, we define a software architecture that turns arbitrary user programs into cloud-ready applications that it then deploys over disparate cloud infrastructures on behalf of user. This software layer, called MEDEA, abstracts away the details of using cloud infrastructures, simplifies program (job) submission, and packages and deploys user programs to any infrastructure that supports background task submission.

The MEDEA front-end consists of scripting language support with which developers describe their programs. Users employ this support to supply meta-information about their programs to the MEDEA execution engine. This metadata includes details about the program (name, executable, arguments, etc.) and the user's account credentials for each cloud they wish to use.

The MEDEA execution engine is a web service that leverages

a commonly available cloud service, the FIFO task queue, to deploy submitted apps as background tasks in different cloud infrastructures. The engine manages jobs in FIFO queues, sets up jobs for execution, persists program output, and collects profile information about the execution. Users access the output and profile information via a web service interface once execution terminates (normally or abnormally).

Moreover, the MEDEA framework is "pluggable" in that each of its components interface to multiple implementations. The components consist of the FIFO queue, the task worker, and storage. In this way, MEDEA programs are portable across any of the component implementations (cloud infrastructure services) that MEDEA plugs in, and users need not become expert with any of the integrated or underlying technologies or modify their applications in any way. In addition, developers can use MEDEA-compatible services to avoid lock-in, to compare and contrast different cloud offerings (their restrictions, costs, performance, etc.), and to evaluate hybrid cloud deployments (cloud interoperability) of their applications, easily and portably.

To implement MEDEA, we leverage the open source AppScale cloud platform [7, 11] and the Neptune HPC configuration language [8]. The plugins that we integrate into the MEDEA deployment engine include compute, storage, and FIFO queue services from Amazon Web Services [1], Microsoft Azure [5], Google App Engine [20], and AppScale. To investigate the potential of MEDEA, we employ the system for a number of different use cases in which we compare and contrast supported plugins in terms of price and performance using various applications, domains, and programming languages.

In the sections that follow, we present the design and implementation of the MEDEA execution model. We describe how we plug in different cloud services with MEDEA. We then investigate the cost and performance of a number of different use cases enabled by MEDEA, empirically evaluate its use in different hybrid cloud configurations, and for programs written in different languages. We then discuss related work and conclude.

3. DESIGN

By unifying cloud program execution under the MEDEA execution model, we aim to make existing user code interoperable between disparate cloud services. Pushing the complexity of cloud services into an abstract software layer reduces the complexity that must be present in user-facing code. This also increases portability, reduces lock-in to a particular vendor's services, and enables users to benchmark their applications without needing to become experts with each technology they wish to utilize. In this work, we focus on providing such support for three different and common cloud services:

- Compute services for execution of user code
- Storage services for data persistence
- Queue services providing a FIFO queue abstraction

The MEDEA execution model uses a combination of these three services to manage and execute programs over supported cloud fabrics. Moreover, it does in a way that hides the details of the implementation of each service, so that users can employ them for execution of their programs without having any knowledge or direct experience with them – users need only have credentials for each cloud she wishes to use.

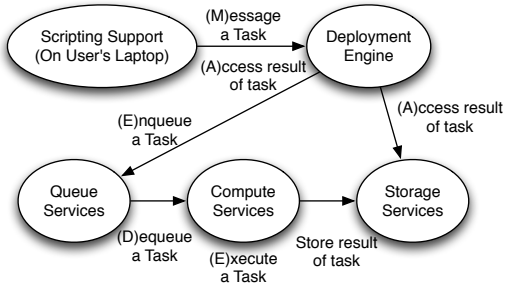

Figure 1: Overview of the design of the MEDEA execution model.

We depict the design of the MEDEA execution model in Figure 1. The MEDEA execution model consists of two components. The first is scripting language support that enables developers to specify the execution environment and deployment preferences for their programs. The second is a deployment engine that plugs in cloud service support to execute applications. We first overview the MEDEA scripting language support and then describe the MEDEA deployment engine.

To make the use of these services portable and simple, the MEDEA scripting support consists of a single function with which users specify the program they wish to run, its inputs, location, and executable (if any), as well as the names of the service plugins they wish to employ for compute, storage, and queuing. The type of programs that MEDEA currently supports are those that take zero or more arguments as inputs, that communicate only with persistent services, and that generate output through the standard output and standard error streams.

Consider a user who wishes to run a Python n-body simulation in Amazon EC2, store its output in Amazon S3 [2], and have workers in EC2 poll for tasks (here, the n-body simulation is the task) via Amazon SQS [4]. Normally they would need to become familiar with the APIs of each service, their pricing models, and best practices. Once this is done, the user is then locked-in to these three services. In contrast, using a domain specific language to specify the execution environment of a program reduces the amount of work required to execute the following code:

```
result = medea(
  :executable => "python",
  :code => "/home/user/nbody.py",
  :compute => "ec2",
  :storage => "s3",
  :queue => "sqs")

puts result.stdout
puts result.stderr
```

Here, the user indicates what binary executes their program in the cloud compute service, where the code to execute is located on their local machine, and which compute, storage, and queue services should be used. Users also provide their credentials to each service as environment variables or as additional parameters. Code can be written in any language, as long as the compute service has the correct binary installed to execute it. Our library support for this function validates the submitter's cloud credentials and verifies that the user's code exists on their local computer. Once this is done, it packages this information and sends it to the MEDEA deployment engine.

The object that is returned from calls to `medea` can be used to manually poll for the result of the job. To execute the program using a different service, the developer need only change the value of a function argument for the compute, storage, and/or queue services. For example, changing the value of `:compute` above from `ec2` to `azure`, causes MEDEA to execute the program in Microsoft Azure instead of Amazon EC2.

The returned object provides methods that store the task's standard output and standard error streams. MEDEA also profiles the execution of the task and returns various performance metrics to the user as a field in this object called `metadata`. This latter support enables users to extend their scripts to interrogate the differences between the multiple cloud services to compare and contrast them and to identify the most appropriate one for their application.

4. IMPLEMENTATION

We implement the MEDEA scripting language support by repurposing Neptune, a domain specific language that automates the configuration and deployment of high-performance computing applications. Our extensions implement this new function (function semantics and library support) to facilitate execution of arbitrary user programs. Users need only specify where the code to execute is located, what executable should be invoked to run it, and an Array of arguments to pass to their program. The scripting language support then validates that the code exists on their local computer, and then copies it and any files specified in the argument list to the MEDEA deployment engine, who then uses its pluggable storage support (Section 4.3) to interface with the storage service that the user has specified.

The MEDEA scripting language support communicates with the MEDEA deployment engine. The deployment engine provides a software layer that abstracts away common services required for execution of arbitrary programs over cloud compute, storage, and queue services. We plug in actual cloud services to this layer to provide the implementation for each of these operations.

The MEDEA deployment engine employs two key abstractions: the Task Manager (which delegates tasks to clouds) and the Task Worker (which executes the task). This scripting language support and deployment engine perform five steps (which form the acronym MEDEA) to execute programs portably:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,
2. the Task Manager (E)nqueues the task to a queue service,
3. a Task Worker (D)equeues a task from a queue service,
4. a Task Worker (E)xecutes the task,
5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script).

We implement the MEDEA deployment engine as a web service within the AppScale cloud platform. This platform automatically deploys and starts the MEDEA deployment engine when an AppScale cloud is instantiated. We also provide plugins into the deployment engine's abstractions for each service (queue, compute, and storage) that implement the necessary functionality.

4.1 Pluggable Queue Support

When a Task Manager receives a request to run a task from a user, it examines the `:queue` parameter in the user's task to determine which cloud queue the task should be placed on. Acceptable values are:

- `"rabbitmq"` for RabbitMQ, hosted within AppScale (the default)
- `"sqs"` for Amazon Simple Queue Service (SQS)
- `"azure-q"` for Microsoft Azure Queue Service
- `"gae-pull-q"` for Google App Engine's pull queue

These queues provide a scalable FIFO queue service where items can be pushed to or popped from. The Task Manager employs the Factory design pattern, thus, as long as supported queues implement a common API (push/pop), the Task Manager can access them without needing to be concerned with their underlying implementation details. Once the Task Manager uses a QueueFactory to get a connection to the necessary queue service, it pushes the task to that queue service and returns an acknowledgement to the user's local computer that the task has been started.

Task Workers periodically query the Task Manager for a list of all the queues that tasks can be found on, as well as the cloud credentials needed to access each queue. This is necessary because two users may have different credentials to the same queue service. Each Task Worker uses the same QueueFactory as the Task Manager to get a connection to each queue service and pops off one item of work per core on its machine.

4.2 Pluggable Compute Support

After pushing the task onto the specified queue service, the Task Manager ensures that Task Workers are running in the specified compute service. For example, if a user has specified that a task should be executed in Amazon EC2, the

Task Manager will ensure that one or more Task Workers are running in Amazon EC2. To provide this functionality, the Task Manager keeps metadata about the number of workers in each cloud and utilizes a ComputeFactory to interact with cloud compute services, based on the value of the `:compute` parameter in the user's task. Acceptable values are:

- `"ec2"` for Amazon EC2, hosted within AppScale (the default)
- `"azure"` for Microsoft Azure
- `"app-engine"` for Google App Engine
- `"euca"` for Eucalyptus

For Amazon EC2, the Task Manager uses the EC2 command-line tools to dynamically spawn or terminate virtual machines. Once virtual machines have been spawned, a Task Worker is started on it, who then polls the Task Manager for work as previously descibed. The Task Manager is also cost-aware, so it does not terminate Task Workers once they have completed a task. Because Amazon EC2 charges on a per-hour basis, the Task Manager terminates Task Workers only near the end of the hour, and only if they are not in use at that time.

Amazon EC2 enables users to remotely log into machines and directly execute programs via the familiar Linux programs `ssh` and `scp`. In contrast, Microsoft Azure and Google App Engine do not support this functionality, as Azure deploys Windows virtual machines, and App Engine does not allow access to the hosted machine at all. To enable interoperable program execution, we contribute `Oration`, a tool that automatically generates Task Workers that execute user-provided applications in different cloud execution systems. Oration takes, as inputs, the name of the cloud to execute the application in, the name of the function to execute, and the name of the file that function can be found in, and then constructs a "cloud-ready" Task Worker that utilizes best practices from that cloud to execute the user's program. This Task Worker implements the following API:

1. `PUT /task`: Given a function name and its inputs, runs the function stores its output for later retrieval.
2. `GET /task`: Given the name of the task, checks to see if the task is still running, has completed, or has failed.
3. `PUT /data`: Given a location to store data and the data to store, saves the data for later use.
4. `GET /data`: Given a location to read from, returns either the given data (if it exists) or a null value (if it does not exist).

For Microsoft Azure, Windows virtual machines are procured (as opposed to Linux virtual machines in Amazon EC2), so the bootup script we include starts by installing language support for each runtime we wish to execute tasks with (by default, this supports Python and Java, but is extensible to other languages). Microsoft Azure also follows a per-hour pricing model, but in contrast to Amazon EC2, it is a per-wall-clock-hour pricing model. The following process is used to implement MEDEA support on Microsoft Azure:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,
2. the Task Manager uses Oration to construct a Microsoft Azure compatible Task Worker and uploads it to Microsoft Azure. The Task Manager then (E)nqueues the task by performing a `PUT /task` on the remotely-hosted web application, which will schedule a background task with the Microsoft Windows Azure Queue Service API. If the task requires any files as inputs, the Task Manager uses `PUT /data` calls to move inputs from the datastore specified to the Windows Azure Storage Service.
3. the Task Worker (D)equeues the task and spawns up workers by performing a `POST /task` to the application server.
4. the Task Worker (E)xecutes the task and stores the output via the Azure Storage Service, a key-value datastore that uses a get/put interface.
5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script). The Task Manager retrieves the result by performing a `GET /data` on the remotely-hosted web application.

Finally, the Google App Engine PaaS provides autoscaling, and does not allow its users to programmatically dictate the number of instances that are used. It also employs a restricted runtime that can only execute tasks written in Python, Java, and Go, so we provide specialized Task Workers in those languages to execute Python, Java, and Go tasks. Google App Engine charges on a per-minute pricing model, as opposed to the per-hour pricing model employed by Amazon EC2 and Microsoft Azure. The following process is used to implement MEDEA support on Google App Engine:

1. the scripting language support (M)essages the Task Manager with the program to execute, described by a MEDEA script,
2. the Task Manager uses Oration to construct a Google App Engine compatible Task Worker and uploads it to Google App Engine. The Task Manager then (E)nqueues the task by performing a `PUT /task` on the remotely-hosted application, which will schedule a background task with the Google App Engine Task Queue API. If the task requires any files as inputs, the Task Manager uses `PUT /data` calls to move inputs from the datastore specified to the Google App Engine Datastore.
3. the Task Worker (D)equeues the task and spawns up workers by performing a `POST /task` to the application server.

4. the Task Worker (E)xecutes the task and stores the output via the Datastore API, an object datastore that uses a get/put interface.

5. the developer (A)ccesses the result of the task from their local computer (from within a MEDEA script). The Task Manager retrieves the result by performing a `GET /data` on the remotely-hosted web application.

Regardless of where the task executes, the Task Worker collects the following data as outputs and metadata from the task:

- The standard output produced by the task.
- The standard error produced by the task.
- The time taken to execute the task.
- The time taken from when the Task Manager received the task to when the task finished executing.
- The time taken to retrieve the code and inputs from the datastore service.
- The time taken to dequeue the task off the queue service.
- Information about the processors on this machine (the contents of `/proc/cpuinfo`).
- Information about memory on this machine (the contents of `/proc/meminfo`).
- Information about disk usage on this machine (the result of `df -h`).

The types of data collected is extensible. In particular, we are looking to extend this system with information about the cost incurred to run the task once cloud providers make this information available programmatically (as opposed to performing estimates or downloading bills from a web page, as is currently done).

4.3 Pluggable Storage Support

Once a Task Worker finishes executing one or more tasks, it uses a StorageFactory to get access to a supported storage service. The user indicates which storage service is to be used via the `:storage` parameter, with acceptable values being:

- `"appdb"` for the datastore hosted within AppScale (the default)
- `"s3"` for Amazon Simple Storage Service (S3)
- `"waz-storage"` for Microsoft Azure Storage Service
- `"gstorage"` for Google Cloud Storage
- `"walrus"` for Eucalyptus Walrus

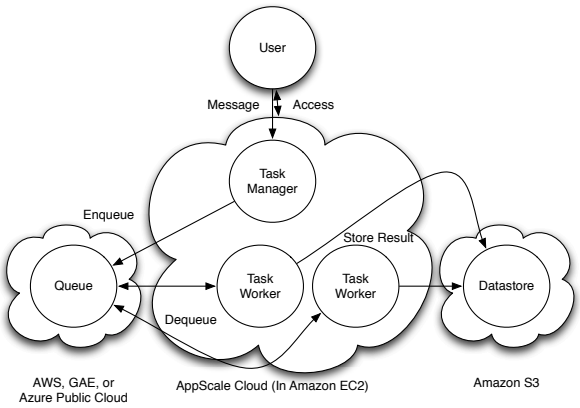

Figure 2: Deployment strategy used for the n-body simulation benchmark to evaluate different pull queue technologies.

The Task Worker then stores three files in the specified storage service, containing the standard output of the task, the standard error of the task, and the task's metadata (performance profile). At this point, if the user's script accesses the `medea` function's return value, the calls will succeed and return this information to the user.

5. EVALUATION

We next use our support for MEDEA within AppScale to empirically evaluate how effectively tasks execute within cloud IaaS and PaaS offerings. We begin by evaluating our pluggable queue support, continue by evaluating a computational systems biology application, and conclude by evaluating implementations of the n-body benchmark application.

5.1 Pluggable Queue Evaluation

We begin by using the pluggable queue support that the MEDEA execution model enables to compare the performance and cost of different cloud queue offerings. We investigate one internal and four external pull queue services: RabbitMQ (internal to AppScale), Amazon SQS, Microsoft Azure Storage Queue, and Google App Engine's pull queue. We employ Amazon S3 as the storage service for each task and deploy an AppScale cloud over Amazon EC2, in the manner shown in Figure 2. Specifically, we instruct AppScale to automatically deploy a single virtual machine instance as the Task Manager, and in all of our Neptune job requests, we indicate that no more than two Task Workers should be dynamically acquired and used (to limit the monetary costs we can incur). The Task Manager creates Task Workers whenever it detects that the number of tasks waiting to be executed in all queues is non-zero. For Task Workers, we utilize Amazon's `m2.4xlarge` instance type, each of which has 8 virtual cores and 68GB of memory. This instance type is one of the more powerful machines offered by Amazon, and costs $1.60 per hour to lease.

For this evaluation, we run ten instances of our n-body simulation program (ten tasks) in parallel and report the time that Task Workers spend dequeuing tasks from the queue. Note that the task's payload is nearly constant for all queues used, and only varies when more or fewer credentials are

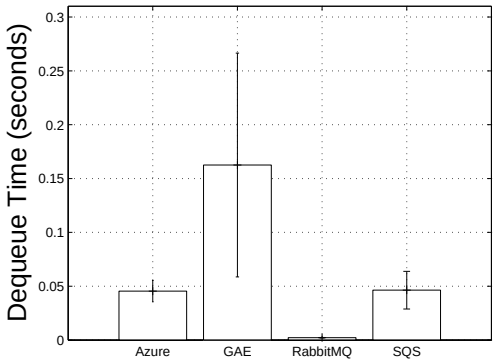

Figure 3: Average dequeue time for the Python n-body simulation, when task data is stored in Azure Storage Queue, Google App Engine's pull queue, RabbitMQ, and SQS. Each value shown here represents the average of ten runs.

needed to access the queue. The Neptune code that we run for each queue to dispatch the tasks and report the resulting time incurred is:

```
task_info = []
10.times { |i|
  task_info << medea(params)
}

task_info.each { |task|
  if task.return_value != 0
    abort(''Analysis failed: '' +
      task.stderr)
  end
  puts task.queue_pop_time()
}
```

The results of running this code for our n-body simulation on each of the four supported queues is shown in Figure 3. RabbitMQ performs the best, because Task Workers in EC2 always have a RabbitMQ server running on their local machine and thus either need only talk to it or to another machine in the AppScale deployment a short distance away. This improves performance but at the cost of fault-tolerance: in the rare case of an availability zone failing in Amazon EC2, it would also cause our RabbitMQ servers to fail with it. Conversely, Amazon SQS and Microsoft Azure Storage Queue have added fault-tolerance, but perform an order of magnitude slower than RabbitMQ, but outperform Google App Engine's pull queue. This is likely due to the latency between our Task Workers and Google App Engine's pull queue.

These results should not be considered a final evaluation of available cloud queuing services, as such services are constantly upgraded and evolve and improve over time. However, since AppScale can be used at any time, users can employ it to snapshot the current performance of the various queue offerings, evaluate that tradeoff against the cost of using the queue, and choose any queue implementation on demand.

5.2 Computational Systems Biology Evaluation

We next evaluate the compute engine offerings that are enabled by making AppScale MEDEA-compatible. The application we use for this study is a Stochastic Simulation Algorithm (SSA) [19]. SSA is form of kinetic Monte Carlo simulation used extensively in computational systems biology. These algorithms are embarrassingly parallel and probabilistic in nature, and require a large number of independent simulations to be executed to achieve an acceptable level of statistical accuracy. The specific algorithm we focus on is the Diffusive Finite State Projection Algorithm (DFSP) [16], which simulates spatially inhomogeneous stochastic biochemical systems. In our study, this algorithm is used to simulate a model of the mating pheromone induced G-protein cycle in budding yeast. We employ this application because it is a canonical example of a compute and data-intensive eScience workflow, thus allowing us to illustrate the performance and cost benefits of executing scientific applications via cloud-based systems. However, it is also an example of an application that is not a web service and thus is not likely to have a user-written MVC interface (which the Task Manager automatically constructs).

Our evaluation considers the Amazon EC2 IaaS, Google App Engine PaaS, and Microsoft Azure IaaS. For the IaaS offerings, we must manually choose the number of instances (virtual machines) that execute tasks, so we experiment with the performance and cost implications of using 1, 2, 4, and 8 workers. For the Google App Engine PaaS, users cannot dictate the exact number of instances to be used (as it dynamically scales up and down in response to user traffic). To provide a fair comparison, we use the `m1.small` instance type within Amazon EC2, the Small instance type within Microsoft Azure, and the `F1` instance type within Google App Engine. These instances minimize the cost incurred to end users, and provide a comparable amount of CPU and memory between one another.

Figure 4 shows the time taken to run a varying number of tasks within Amazon EC2 and Microsoft Azure, for varying numbers of workers. As we increase the number of simulations, we see a roughly linear increase in the amount of time taken to execute these tasks. We also note a standard deviation proportional to the number of tasks run. For Amazon EC2, this is due to the performance variability of tasks that execute within it, a result that has been confirmed by the works of others [6] [25]. As we increase the number of workers used to execute tasks, we also note a corresponding speedup in the total execution time. Note that the x-axis is on a logarithmic scale.

Figure 5 shows the time taken and the cost incurred when using the maximum number of workers in Amazon EC2 and Microsoft Azure (here, 8 workers) and compares it with Google App Engine, which autoscales and does not allow us to dictate the exact number of workers to use. We note that Google App Engine performs the best of the three engines compared here, and because of its per-minute pricing model, costs less than the other offerings for the lower numbers of simulations. Amazon EC2 and Microsoft Azure both cost the same, which is simply the price of eight machines for a single hour. All three cost a similar amount when the

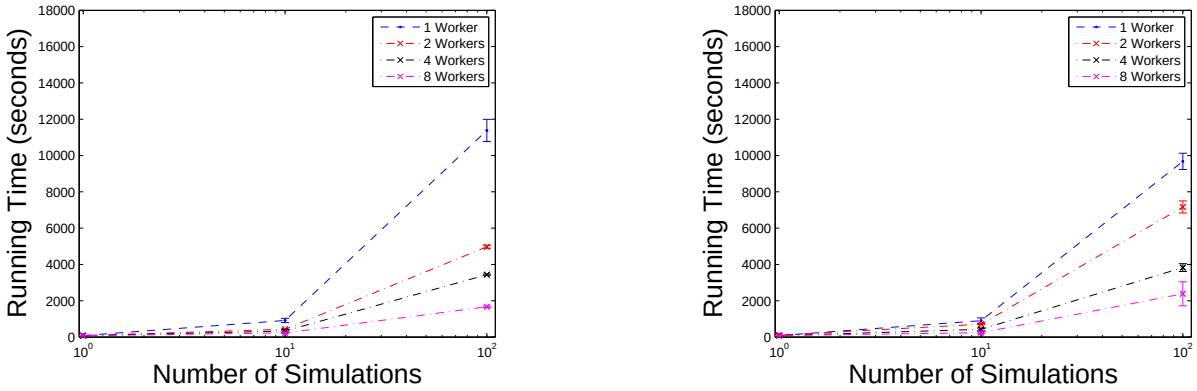

Figure 4: Running time for execution of SSA simulations in Amazon EC2 (left) and Microsoft Azure (right), when a varying number of workers are utilized. Each value represents the average of five runs. Note that the x-axis is on a logarithmic scale.

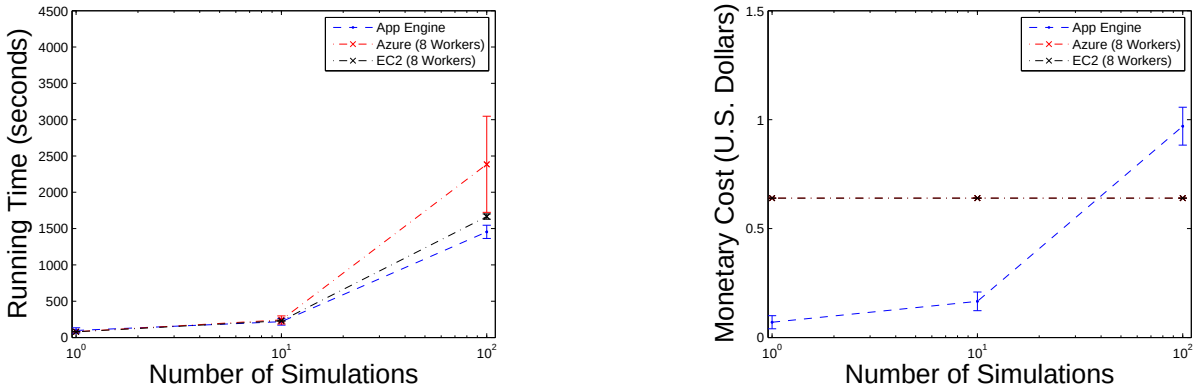

Figure 5: Running time (left) and monetary cost incurred (right) for execution of a varying number of SSA simulations in Amazon EC2, Microsoft Azure, and Google App Engine. Each value represents the average of five runs.

total execution time approaches an hour, which agrees with the per-hour pricing model employed by Amazon EC2 and Microsoft Azure.

5.3 Programming Language Shootout Evaluation

In the previous sections, we showed that the MEDEA execution model can be used to enable programs to be executed simply and easily over disparate cloud systems. In this section, we use AppScale's MEDEA support to compare the performance and cost of using different programming language implementations of programs over different public cloud fabrics.

It can be useful to test the performance of a given language, which itself evolves into numerous versions over time. Additionally, creators of a new programming language may wish to compare the performance of their language with other programming languages on a set of reference implementations. In the spirit of the Computer Language Benchmarks Game [30], we can use AppScale (augmented with MEDEA) to provide a community cloud PaaS that can be used to benchmark algorithms with implementations in different languages on various cloud compute, storage, and queue services.

We evaluate AppScale's MEDEA support in this use case in Figure 6. Here, we have taken eleven implementations of the n-body simulation benchmark from [30], written in programming languages of varying programming paradigms, type checking systems, and other language-level design and implementation details. This data shows that most of the implementations of this benchmark perform within the same order of magnitude, with the exceptions of Python and Ruby, which perform two orders of magnitude slower than the others. These results are roughly in agreement with the values published by [30].

While the MEDEA execution model provides users with support for different programming languages and different programming models, it also enables users to investigate and understand the monetary costs of using a particular programming language in a public cloud setting. Moreover, it enables users to investigate the costs of the different pricing models employed by public cloud vendors.

For example, the cost to run the n-body benchmark in different languages using AppScale over Amazon EC2 is shown in Table 1. We consider both the cost to run each benchmark via an hourly pricing model (the standard employed by Amazon) and a per-second pricing model (similar to the per-

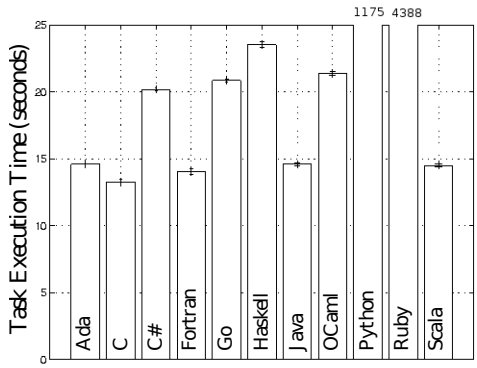

Figure 6: Average running time for implementations of the n-body benchmark in different programming languages. Only the time taken to execute the task is considered here. This does not include the time taken to message the system, enqueue the task, dequeue it, or the final result in the remote datastore. Each value represents the average over ten runs.

Language	Cost Per Task
Ada	\$0.0076 ± \$0.0002
C	\$0.0069 ± \$0.0002
C#	\$0.0105 ± \$0.0000
Fortran	\$0.0073 ± \$0.0003
Go	\$0.0105 ± \$0.0000
Haskell	\$0.0120 ± \$0.0000
Java	\$0.0075 ± \$0.0000
OCaml	\$0.0110 ± \$0.0000
Python	\$0.5876 ± \$0.0057
Ruby	\$2.1944 ± \$0.0198
Scala	\$0.0075 ± \$0.0000

Table 1: Average monetary cost (in U.S. dollars) incurred to run the benchmarks shown in Figure 6 via a per-second pricing model. These costs only include the cost incurred for the virtual machines used. Each value shown here represents the average cost incurred over ten runs.

minute pricing model employed by Google App Engine). For the hourly pricing model, all of the benchmarks employed ran within a single hour (except for Ruby), and thus cost \$1.80 to run. For Ruby, it took more than an hour to run, so we were assessed charges for two hours of computation, a total of \$3.60. If Amazon were to employ a per-second pricing model (as shown in the table), the results exhibit larger differences between language technologies. Specifically, C is the cheapest, with Fortran, Java, Scala, and Ada closely following it. Python and Ruby perform the slowest, costing one to two orders of magnitude more to run.

AppScale, with MEDEA support, thus provides users with a tool that they can use to measure the costs of running their application in a given language or under the different pricing models employed by cloud vendors. Such a tool is important for the investigation of new pricing models and to assess application costs when pricing models change.

Next, we consider the performance and cost of running the Python and Java n-body simulations in Amazon EC2, Google App Engine, and Microsoft Azure. We elect to use only Python and Java (as opposed to all the languages we have implementations for) because Google App Engine only supports programs written in Python, Java, and Go. Here, we utilize a `m1.large` instance in Amazon EC2, a `F4` instance in Google App Engine, and an `Extra Small` instance in Microsoft Azure. We vary the number of bodies to simulate between 5×10^3 and 5×10^7, and run each simulation ten times, reporting the average and standard deviation.

The average running time for the n-body simulation benchmark is shown in Figure 7. Amazon EC2 performs the fastest at the lower number of bodies to simulate because it does not dispatch workers to a queue and backend storage service - it simply runs them as it receives them. At the higher number of bodies to simulate, the queue and storage service time no longer dominates the total execution time, and the three services perform roughly the same to one another. We were unable to run the Python n-body simulation at 5×10^7 bodies, because our instances used more than 512MB of memory (the maximum memory allowed for `F4` instances) and were killed by the App Engine runtime. Even without this memory restriction, it would have likely taken more than 10 minutes to execute (the maximum time allowed for background tasks to execute within Google App Engine) and still have been killed by the Google App Engine runtime.

The average cost to run the n-body simulation benchmark is shown in Table 2. Amazon EC2 and Microsoft Azure charge users on a per-hour basis, and because all of the n-body simulation times for the Python and Java implementations ran in less than an hour, we were charged for a full hour in these systems. This was \$0.32 for a `m1.large` instance in Amazon EC2, and \$0.02 for an `Extra Small` instance in Microsoft Azure. Google App Engine charges on a per-minute basis, and because all of the Java n-body simulation times ran in less than a minute, we were charged for a full minute in Google App Engine (as opposed to a full hour in Amazon EC2 and Microsoft Azure). As we used the most expensive instance type in Google App Engine (the `F4` instance type), we were charged \$0.0013 for the minute that our program took to execute. We used the same instance type for our Python n-body simulation, but as the larger number of bodies to simulate took more than a single minute to execute, we were charged for more than a single minute of time. Table 2 shows the cost incurred by simulating 5×10^7 bodies.

These cost values are not intended to reflect the optimal costs of running the n-body simulation code in Amazon EC2, Google App Engine, and Microsoft Azure. We could have picked instance types that cost less within each of these providers, which could have then increased the total execution time for each, which could have then increased the cost incurred (depending on the pricing model used). As was the case for the cloud queue services, implementations of MEDEA provides users with a system that can be used to snapshot the performance and cost of using a cloud IaaS or PaaS system to execute their code.

6. EXTENDING MEDEA

The MEDEA execution model enables cloud interoperability for supported programs. In this section, we consider exten-

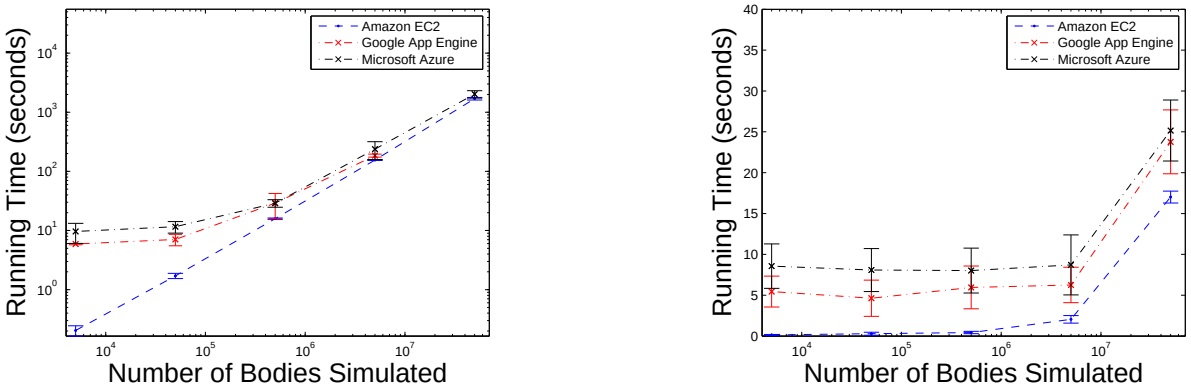

Figure 7: Running time for execution of n-body simulations written in Python (left) and Java (right), using Amazon EC2, Google App Engine, and Microsoft Azure. Note that both axes are on logarithmic scales.

Cloud Service	Cost
Amazon EC2	$0.3200 ± $0.0000
Google App Engine (Java)	$0.0013 ± $0.0000
Google App Engine (Python)	$0.0049 ± $0.0006
Microsoft Azure Worker Roles	$0.0200 ± $0.0000

Table 2: Monetary cost incurred to run the n-body simulation code shown in Figure 7 across Amazon EC2, Google App Engine, and Microsoft Azure. Costs are assessed on a per-hour basis for Amazon EC2 and Microsoft Azure, and on a per-minute basis for Google App Engine. The value presented for the Python Google App Engine simulation reflects only the most expensive simulation size (all others are identical to the Java Google App Engine simulation).

sions to this model to facilitate greater portability across and ease of use of cloud systems. The extensions we consider in the subsections that follow include simplifying the use of the MEDEA scripting language component via a parallel future construct, task inlining (bypassing the queuing system in some cases), making task deployment more efficient via batching, and utilizing caching on Task Workers to eliminate unnecessary data retrieval. Throughout this section, we also consider the impact of these optimizations on the popular MapReduce programming model, popularized in [13], with an emphasis on the single embarrassingly parallel applications that this model supports.

6.1 Automatic Polling via Futures

We begin by considering ways to improve the use of the MEDEA scripting language component for distributed, multi-cloud application deployment. Towards this end, the result of an invocation of `medea()` returns an object that encapsulates information about the task's execution. Users can poll for the output of the task from within a MEDEA script, to determine when a task has completed. The MEDEA script for doing so would look similar to the following:

```
result = medea(params)

output_params = params.dup
output_params[:type] = "output"
loop {
  if medea(output_params)[:done]
    break
  end
  sleep(10)
}

puts result.stdout
puts result.stderr
puts result.metadata
```

To automate the process of polling (reducing the amount of work a user must perform) and to enable the script to do other work while waiting for the task to finish (e.g. execute more tasks), we investigate implementing the `medea` function as a future [22, 32, 31]. A future is a simple and elegant programming language construct that enables developers to introduce asynchronous computation into their programs.

To enable this in our scripting language support, we modify the design and implementation of the `medea` function to return a future for the object (the task's result) instead of the object itself. When `medea` is invoked, a background thread is spawned that dispatches a message to the MEDEA Task Manager, polls for its output, and blocks if the user calls any of its methods or accesses any of its fields before the task has completed. We employ Ruby's metaprogramming features to implement implicit future semantics for the `medea` function, so users need not know that the object they are accessing is a future. The previous example, which used polling, can be rewritten when futures are used, as follows:

```
result = medea(params)
puts result.stdout  # this will block until
puts result.stderr  # the task completes
puts result.metadata
```

6.2 Inlining Task Execution

MEDEA provides a task execution model that utilizes a distributed queue service to pass information between the Task Manager and Task Workers. Yet for short-running tasks, the overhead incurred by storing tasks in a queue may be longer than running the task immediately within the Task Manager. Therefore, we enable users to specify the value of `:worker` to be `inline` to indicate that the task should be "in-

lined" - that is, it should not follow the standard MEDEA execution model, and instead should be immediately executed inline within the Task Manager.

To evaluate the benefits and drawbacks of task inlining in MEDEA, we deploy a set of tasks that count the number of words in an input corpus using the map-reduce programming model [13]. Here, each Map task performs a word count on the works of William Shakespeare (roughly 5MB in size), and each Reduce task aggregates the results from each Map task. Our extensions to the MEDEA scripting language support that facilitate the use of futures enables supported programs to be "chained" together in a manner similar to that of a workflow system, except that this system is fully Turing-complete, as opposed to the XML-based systems that most workflow systems employ. Here, we pass the output of each Map task as an input to the final Reduce task. The MEDEA script for this MapReduce job looks like the following:

```
common_params = {
    :storage => "s3",
    :queue => "sqs",
    :instance_type => "m2.4xlarge",
    :max_nodes => 3,
    :worker => "inline"
}

map_params = common_params.dup
map_params[:code] = "/.../wc.py"
map_params[:argv] = ["/.../shakespeare.txt"]

num_mappers.times { |i|
  param_list << map_params
}
map_tasks = medea(param_list)

outputs = map_tasks.map { |task|
  task.output_location
}
reduce_params = common_params.dup
reduce_params[:code] = "/.../reduce.py"
reduce_params[:argv] = outputs
reduce_task = medea(reduce_params)
```

In this experiment, we vary the number of Map tasks dispatched when inlining is used and when it is not used, and report the results in Figure 8. The data shows that when we inline a small number of tasks, inlining performs better than the non-inlined case, but as we inline more tasks, it causes a near-linear slowdown on the system (as all inlined tasks are run on the Task Manager, who runs them serially). In the non-inlined case, the number of tasks we run are smaller than the number of available cores, so the total execution time is roughly constant.

As part of ongoing and future work, we are investigating how to automatically detect when a task can and should be inlined vs deployed via the tasking system. Such support will remove the burden from the programmer to decide when it is best to do so. Since the MEDEA Task Manager collects performance data and task behavior, we will use this information to guide this inlining functionality that we currently

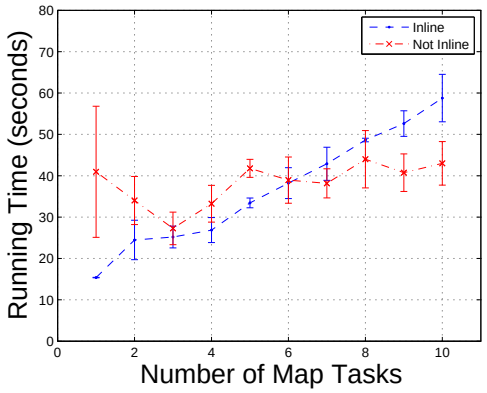

Figure 8: Average end-to-end time to run inlined and non-inlined tasks for the Python MapReduce WordCount code for varying numbers of Map tasks and a single Reduce task. Each value here represents the average of five runs.

have in place. As a first step, we augment the Task Manager to automatically inline up to one task per core on its node (to avoid CPU thrashing from overprovisioning tasks).

6.3 Batch Task Execution

As many real-world use cases need to run more than a single task, the ability to batch task invocations can be useful. Our next MEDEA extension therefore facilitates batch task invocation.

To enable this, we modify the invocation of `medea` to take advantage of Ruby's duck typing capabilities so that it can receive either a Ruby hash (a single task invocation) or a Ruby array of hashes (multiple task invocations) as arguments. In case of the latter, the multiple task requests are dispatched all at once to the MEDEA Task Manager within AppScale, and a Ruby array of futures of task objects is returned as a result. A code example that runs ten n-body simulations in Google App Engine and prints their outputs is:

```
tasks = []
10.times { |i|
  tasks << medea(params)
}

tasks.each { |task|
  puts task.stdout
}
```

That example dispatches 10 tasks individually to MEDEA to be executed, and prints the result of each task. Alternatively, the 10 tasks could be dispatched in a single batch request as follows:

```
param_list = []
10.times { |i|
  param_list << params
}

tasks = medea(param_list)
```

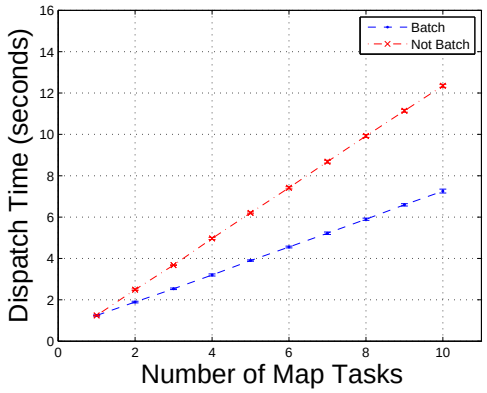

Figure 9: Average time to dispatch requests in a batched fashion and a non-batched fashion for the Python MapReduce WordCount code for varying numbers of Map tasks. Each value here represents the average of ten runs.

```
tasks.each { |task|
  puts task.stdout
}
```

Figure 9 shows the performance improvements that are possible from batching requests for the Python MapReduce WordCount code. When only a single Map task is used, the two systems perform roughly equally. However, as the number of tasks to run increases, batching the tasks into a single request saves a significant amount of time. The amount of time spent is linear in the number of tasks in both cases, as the MEDEA scripting language checks that the inputs and code to run are in the remote datastore (or copy them to the datastore if they are on the local disk), and that the output location specified does not exist (to avoid accidentally overwriting existing data).

6.4 Caching Support

Many use cases, such as those in the MapReduce programming paradigm, can execute many instances of a single program (here, the Map program) on a single machine. Our final MEDEA extension is therefore concerned with providing caching for programs and inputs on machines.

To implement caching support, whenever Task Workers would normally download a program or an input file, they first check to see if they have the file already stored locally in `/var/cache/medea`. If so, they do not attempt to download the file again (otherwise, they download the file from the remote datastore as usual).

Figure 10 shows the results of executing WordCount Map tasks over the baseline MEDEA system, as well as the performance improvements that occur when we batch the tasks into a single request. Finally, we also consider the performance improvements of using batching as well as caching the Map program and its input file (the works of William Shakespeare). Although batching does improve performance (by 23% at 64 Map tasks), adding caching support has a much greater impact on total execution time (by 65% at 64 Map tasks). This is because the input file is 5MB in size, so not

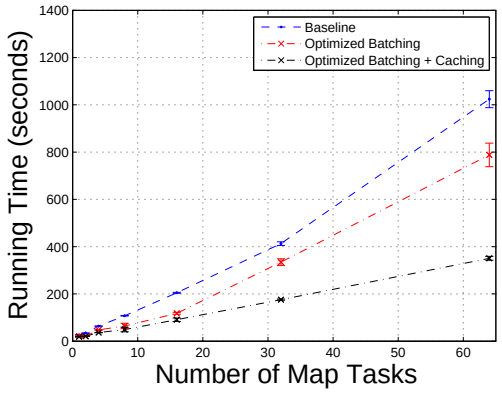

Figure 10: Average time taken to execute a varying number of WordCount Map tasks, when the baseline system is used, when batch task support is enabled, and when batch task support and caching is employed. Each value here represents the average of five runs.

re-downloading it for every Map task reduces the normal "download-execute" process to simply "execute".

7. RELATED WORK

This paper is an extended version of a previous version of this work that is currently in submission [**?**]. In this paper, we detail the design of the system and describe the benefits that MEDEA facilitates in terms of cloud interoperability through application portability and deployment automation across cloud fabrics. In addition, we evaluate a number of different extensions and use cases (cf Section 6) not included in the earlier work.

Our contributions in this paper build upon and extend the work of others. In particular, we repurpose the AppScale cloud platform and the Neptune cloud configuration language to facilitate MEDEA implementation. AppScale is an open source distributed runtime that deploys, executes, and scales Google App Engine applications written in Python, Java, and Go. Neptune is scripting language support that enables HPC applications written in MPI [21], UPC [17], X10 [10], KDT [26], and others to be deployed over AppScale. MEDEA extends this work by providing automatic execution of programs written in any programming language, across compute, storage, and queue services offered by Amazon, Google, and Microsoft. Furthermore, our extensions automatically collect and expose metadata about the program, allowing users to write programs that quantify the performance characteristics of the programs they execute.

MEDEA is inspired in part by the YCSB project [12] and its successor, YCSB++ [28]. These projects enable users to benchmark popular non-relational datastores (e.g., HBase [23], Cassandra [9]) on a consistent workload to provide information about their underlying performance characteristics. MEDEA goes a step in an orthogonal direction: instead of providing a system that can be used to benchmark datastores in a single cloud IaaS, implementations of MEDEA can be used to benchmark compute, storage, and queue services tied together in a single cloud IaaS or PaaS, or utilized as a hybrid cloud.

Elastisizer [24] provides users with the ability to automatically acquire IaaS resources and run tasks over them, and like Neptune, provides a language-like interface to abstract away resource usage. Elastisizer differs from MEDEA in two critical ways. First, Elastisizer can run only Java tasks that conform to the Hadoop MapReduce framework / programming model, whereas our implementation of MEDEA can run tasks written in any programming language, in any programming model. Secondly, Elastisizer's declarative language serves a different purpose than Neptune does. Elastisizer enables users to query the system about the performance of their tasks for certain data sets, while Neptune enables users to specify the tasks themselves and chain them together with other tasks.

[18] attempts to solve a similar, but in many ways an orthogonal, problem. While MEDEA aims to simplify application deployment over disparate cloud resources, [18] aims to optimize application scheduling for different types of resources within a single cloud. In particular, [18] focuses on the Amazon EC2 public cloud, and constrains itself to that set of APIs. Conversely, MEDEA seeks to maximize the APIs that it supports, and does not explicitly optimize application scheduling within cloud services (although it could be extended to do so).

In a similar vein, Pegasus [14] and Swift [15] allow users to specify an execution plan (typically in XML) to connect programs together. In contrast to our language support, these execution plans are not Turing-complete, which prevents them from being used in scenarios where the result of a computation can cause an arbitrary piece of code to be executed or require some type of human interaction (which may be the case when an expert user is needed to analyze the result of a computation). Furthermore, these systems are not designed to be pluggable in nature: they intend only to utilize a single, statically owned set of resources to run applications.

Workflow systems execute and connect programs together automatically, which is conceptually similar to what MEDEA offers. AME [33], Condor [29], StratUm [27], and Amazon Simple Workflow Service (SWF) [3] are recent works that seek to address this problem, for differing domains. AME is designed to run on supercomputers, where millions of cores may be present, while Condor and StratUm utilize grids, which do not provide elasticity and thus do not allow users to dynamically acquire nodes. While Amazon SWF does operate within a cloud environment, it is specialized to the Amazon cloud, which encourages lock-in to Amazon's compute, storage, and queue services. Furthermore, the specification language that connects computation together in Amazon SWF is not Turing-complete, limiting the types of computation that can be run in a manner similar to Pegasus and Swift.

8. CONCLUSION

We contribute MEDEA, an execution model whose implementation automatically deploys user programs to cloud IaaS and PaaS systems (compute, storage, and queue services), without requiring that users modify their applications. To provide this pluggable service, MEDEA repurposes an open source cloud PaaS and domain specific language to enable arbitrary programs to be deployed and executed. Our implementation of MEDEA encapsulates such programs automatically so that they can be executed over a wide variety of cloud systems, and can execute programs on-premise or off-premise in Amazon EC2, Google App Engine, Microsoft Azure, or some combination.

We experiment with and evaluate MEDEA's implementation using a number of different programs, programming languages, benchmarks, and use cases. We find that while cloud systems may perform similarly for a given piece of code, they can vary greatly with respect to the price users pay to run their code in these systems, due to the pricing models that clouds enforce. Overall, the MEDEA execution model significantly simplifies and makes cloud IaaS and PaaS systems portable and reusable through abstraction and a cloud PaaS system. With our implementation of MEDEA, users can "snapshot" the performance and cost of their programs in cloud systems, and run them where it is fastest or cheapest to do so. All of the work presented in this paper has been committed back to the AppScale and Neptune projects under a permissive open source license, whose source code can be found at `http://appscale.cs.ucsb.edu`.

9. REFERENCES

[1] Amazon Web Services. `http://aws.amazon.com/`.

[2] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`.

[3] Amazon Simple Workflow Service (SWF). `http://aws.amazon.com/swf`.

[4] Amazon Simple Queue Service (Amazon SQS). `http://aws.amazon.com/sqs/`.

[5] Microsoft Azure Service Platform. `http://www.microsoft.com/azure/`.

[6] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. *MMsys*, pages 35–46, 2010.

[7] 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.

[8] C. Bunch, B. Drawert, N. Chohan, C. Krintz, L. Petzold, and K. Shams. Language and runtime support for automatic configuration and deployment of scientific computing software over cloud fabrics. *Journal of Grid Computing*, 10:23–46, 2012. 10.1007/s10723-012-9213-8.

[9] Cassandra. `http://incubator.apache.org/cassandra/`.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[11] 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.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st*

ACM symposium on Cloud computing, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

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

[14] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus : Mapping Scientific Workflows onto the Grid. In *Across Grids Conference*, 2004.

[15] J. Dias, E. Ogasawara, D. de Oliveira, F. Porto, A. L. Coutinho, and M. Mattoso. Supporting dynamic parameter sweep in adaptive and user-steered workflow. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 31–36, New York, NY, USA, 2011. ACM.

[16] 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.*, 132(7), 2010.

[17] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[18] M. R. H. Farahabay, Y. C. Lee, X. Liu, H. R. Dehkordi, and A. Y. Zomaya. Technical Report TR-684, University of Sydney, 2011.

[19] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[20] Google App Engine. `http://code.google.com/appengine/`.

[21] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[22] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[23] HBase. `http://hadoop.apache.org/hbase/`.

[24] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[25] Y. E. Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the performance fluctuations of hpc workloads on clouds. *CloudCom*, pages 383–387, 2010.

[26] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SIAM Conference on Data Mining (SDM)*, 2012 (accepted).

[27] P.-O. östberg, A. Hellander, B. Drawert, E. Elmroth, S. Holmgren, and L. Petzold. Abstractions for scaling escience applications to distributed computing environments; a stratum integration case study in molecular systems biology. *Proceedings of BIOINFORMATICS 2012, International Conference on Bioinformatics Models, Methods, and Algorithms*, pages 290–294, 2012.

[28] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[29] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[30] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

[31] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in java. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 175–184, New York, NY, USA, 2007. ACM.

[33] Z. Zhang, D. S. Katz, M. Ripeanu, M. Wilde, and I. T. Foster. Ame: an anyscale many-task computing engine. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 137–146, New York, NY, USA, 2011. ACM.