Bridging the Gap between Desktop and the Cloud for eScience Applications

Yogesh Simmhan, Catharine van Ingen
Microsoft Research
Los Angeles, CA
{yoges,vaningen}@microsoft.com

Girish Subramanian
Indiana University,
Bloomington, IN
subramag@cs.indiana.edu

Jie Li
University of Virginia,
Charlottesville, VA
jl3yh@virginia.edu

The widely discussed scientific data deluge creates a need to computationally scale out eScience applications beyond the local desktop and cope with variable loads over time. Cloud computing offers a scalable, economic, on-demand model well matched to these needs. Yet cloud computing creates gaps that must be crossed to move existing science applications to the cloud. In this article, we propose a Generic Worker framework *to deploy and invoke science applications in the cloud with minimal user effort and predictable cost-effective performance. Our framework addresses three distinct challenges posed by the cloud: the complexity of application deployment, invocation of cloud applications from desktop clients, and efficient transparent data transfers across desktop and the cloud. We present an implementation of the Generic Worker for the* Microsoft Azure Cloud *and evaluate its use for a genomics application. Our evaluation shows that the user complexity to port and scale the application is substantially reduced while introducing a negligible performance overhead of < 5% for the genomics application when scaling to 20 VM instances.*

1. Introduction

Cloud computing has emerged as a viable platform for running large-scale computation and data analysis [8]. Building on prior work on Grid and cluster computing, it offers an evolutionary paradigm that approaches utility computing [3,11,13]. The advantages of cloud computing are well known: on-demand and scalable resources, pay as you go policy that avoids costly capital costs, competitive pricing due to economies of scale, and simple service interfaces for easy access and management. Commercial Clouds backed by large global datacenters have been available for some time from vendors such as Amazon [24], RackSpace [23], and Microsoft [22]. There is also active research on open source and private clouds [14], such as Eucalyptus [4], Nimbus [30] and CAIRN [12], and on improving cloud operations [6,18].

Cloud computing holds benefits for science applications at different scales [7,16,29], from the desktop to the supercomputer. Desktop applications that have outgrown single machine cores can leverage the cloud on demand rather than build and maintain clusters [10]. Existing applications on older clusters can migrate to cloud virtual machines (VMs) that track technology to provide a quicker and cheaper alternative to cluster upgrades or securing access to supercomputing centers [5]. Supercomputing users can consider the cloud for surge computing during peak loads to lower queue latency [3,9].

While cloud computing is promising for scaling eScience applications beyond the desktop, it poses significant challenges for many users [20,21,29]. In this article, we focus on tools to develop, deploy, and operate eScience applications on commercial clouds. Specifically, we identify and address three barriers to wider eScience usage of cloud:

1. *Application Deployment* – Ease of application porting and initial deployment to the cloud,
2. *Application Execution* – Simple invocation and tracking of cloud applications from remote clients,
3. *Data Access* – Efficient data access for eScience applications across the desktop and the cloud.

Our *Generic Worker framework* lowers these barriers by providing a set of intuitive desktop client tools and a single wrapper binary that executes on all cloud VMs. The framework supports simple registration and deployment of .NET, Java, and command line applications. Once deployed, cloud applications can be instantiated on-demand and at scale from a desktop or remote client: through a command shell, from a workflow, or a .NET or web service API. The framework tracks runtime execution and data provenance automatically. Lastly, the framework transparently performs automatic and just in time file movement between the desktop, cloud storage and cloud VMs with efficient caching.

We implement the framework on Microsoft's Azure Cloud platform [22] and evaluate its ease of use and performance for a real genomics application. While our implementation is on Azure, we believe the same Generic Worker pattern applies to Amazon EC2 and other IaaS/PaaS clouds [31], albeit with somewhat different implementation.

The rest of the paper is structured as follows. In Section 2, we discuss the challenges of migrating eScience applications to the cloud and related work. Section 3 introduces the Generic Worker framework. Section 4 evaluates the framework used for a genomics application. Lastly, Section 5 presents conclusions and future work.

2. Motivation and Related Work

Cloud computing is commonly categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [31]. IaaS, such as Amazon EC2, abstracts the hardware resources as a virtual machine to give the equivalent of bare metal machine. PaaS, such as Google AppEngine, provides a scalable programming platform that runs within a sandbox. SaaS provides applications as scalable services that can be customized or composed into other applications, examples being business applications from SalesForce.com. This paper focuses on effective use of IaaS and PaaS clouds for scientific applications; eScience SaaS will arrive when these applications become offered as public services.

2.1. Cloud Computing Benefits for eScience

Cloud computing offers benefits [7,16,29] for eScience applications at scales ranging from desktop scientists to supercomputing power users.

Desktop applications must be scaled out rather than simply scaled up because scientific computing and data needs are outpacing Moore’s Law and local disk capabilities [32]. The prospect of building, operating, and porting applications to a cluster can be daunting as scientists must rewrite code and become cluster operators. The cloud can offer longer-term scalability and hide much of that complexity if programming models and interfaces remain simple [33]. Also, the cyclic nature of some scientific computing needs, such as just after annual field campaigns, fits well with the on-demand flexibility offered by the cloud [32].

For existing *local research cluster* users, the cloud can reduce the lower real (non-subsidized) total cost of ownership by reducing the operations people cost, peak networking bandwidth requirements, and technology upkeep through constant machine upgrades [5]. For cluster users considering a step up to national supercomputing centers, paperwork and eligibility requirements may restrict and delay access as well as limit the software stack or services that can be deployed due to security restrictions. With cloud computing, all it takes is a credit card to open a new account online and access compute node VMs.

Users of *supercomputing centers* can use the near-instantaneous access of clouds to reduce the queue wait times [9]. Clouds can also be used as resource surge during peak load at centers, or during special events like tutorials that require large, reserved resources for guest users for a short duration [3]. Lastly, large datasets generated in the cloud can be easily shared for analysis within or outside the cloud, democratizing access to scientific results. Supercomputing centers restrict *ad hoc* hosting of datasets and limit their local analysis to authorized users.

2.2. Migrating Applications to the Cloud

Running an eScience application in an infrastructure cloud presents three different challenges: (1) application deployment, (2) application execution, and (3) file transfers to, from, and within the cloud.

2.2.1. Application Deployment

Simplistically, deploying an application to an IaaS cloud requires installing it on an existing or new VM image and then copying the image into cloud storage so instances of that VM can be created [25]. This model is used in Amazon EC2. Deployment on the Microsoft Azure PaaS is similar. Each application must be packaged as a .NET “role”. That package is then copied into cloud storage so that instances of that role can be created by copy to a Windows 64bit Server VM.

Both clouds require the scientist user to manage application binaries and dependencies [29]. Images and packages must be recreated and redeployed in the event of any change to an application, its configuration or dependencies [20,21]. For example, the user must explicitly manage application dependencies on a version of MatLab and the Java VM. This becomes more complicated when several applications are deployed in a single VM for optimal use of VM resources. Users must either maintain many VM images and running instances, or stage application dependencies on the fly within a transient VM instance.

2.2.2. Application Execution

Applications deployed in cloud VM instances need to be invoked from outside that instance – from workflows, scripts, command lines or web forms executing in or outside the cloud. This requires the application and its clients to be modified to support remote calls and/or a custom proxy service written that marshals and unmarshals parameters as requests passed as service calls or messages in reliable cloud queues. Code maintenance becomes tedious as applications and their clients increase in time or their APIs change.

2.2.3. Data Transfer

Passing files as parameters to cloud applications introduces problems in managing data transfers between clients outside the cloud and VM instances [3]. Files on local disks or network shares are important, often primary, inputs and outputs for scientific applications. Cloud applications need to access files passed as input by clients outside the cloud, and vice versa for output files. For example, a desktop workflow may use applications on the local machine and the cloud with files passed between them. Also, since VM local disks are transient, any persisted files must be moved to cloud persistent storage. Not all cloud platforms provide a network file system accessible both inside and outside the cloud. This necessitates data transfers using REST protocols to cloud storage that are less intuitive for scientific applications and require them to be changed to use such protocols. Lastly, the low bandwidth into and out of the cloud means users have to be more aware of client data transfers.

2.3. **Related Work**

Cloud vendors provide tools to install and manage applications in a VM image that is then copied to and scaled in the cloud. Examples include Amazon EC2 AMI tools [24,25] and image managers like OpenNebula [27]. However, a VM image is too coarse a granularity to manage for a constantly changing pool of applications where the upload time can add significant delay – a 2GB VM image takes approximately 15mins to upload [26]. Also, these tools do not address the challenge of remote execution or efficient data transfer between desktop and cloud.

A profile-based approach proposed recently [35] allows on-demand deployment and load-driven scale out of applications onto clouds. While this partly addresses the dynamic deployment problem we solve, it requires install scripts configured by experts to update predefined VM images. Also, several persistent infrastructure services are required for the architecture, relegating its use to large institutions and advanced users. In comparison, our single-step application registration process is more accessible to most eScience users and the single, self-contained Generic Worker VM (that also executes the applications) is the only custom service required.

Similar application and service deployment tools [18] for clusters and grids, such as ADEM [28], GADe [15] and Distributed Ant [19], support complex dependencies, but are meant for online nodes rather than VM images. Using them with transient VM instances induces repeated deployments. Also, their complexity is prohibitive for scientists wishing to deploy and run applications that have a self-contained set of binaries and libraries in the cloud.

Toolkits such as GFac [1], Opal [2] and gRAVI [36] use a factory pattern to create dynamic web services that wrap commandline applications on the grid. They allow users to describe the commandline parameters and create services with matching interfaces that when invoked, schedule the application to execute using a batch scheduler. While the intent of easily supporting remote application is similar, they differ from our Generic Worker model in several ways:

1. The toolkits require users to explicitly specify parameter bindings. We additionally support automatic binding of .NET or Java applications using reflection.
2. Only gRAVI packages applications for automatic deployment like we do instead of assuming the applications are already installed.
3. The toolkits do not address data transfer optimization. Our file caching and just in time data movement avoids costly file movement between cloud and client unless required.
4. All of these toolkits are tightly coupled to grid installations for job scheduling and file transfer using GridFTP. We instead focus on cloud platforms and reuse simple native cloud primitives like reliable queues and REST storage protocols. This makes our framework readily usable without complex installation and more easily portable across cloud vendors.

Programming platforms such as workflow systems [34], Dryad [37] and Hadoop [38] automate application and file transfer across heterogeneous environments. However, they require applications to be composed as workflows or Map-Reduce constructs. Our model is an *execution platform*, which allows application deployment without modification by supporting application execution and transparent data transfer through command shell, APIs or even workflows.

3. **Generic Workers for Cloud Applications**

In this section, we describe the architecture of our Generic Worker framework. A Generic Worker is an instance of our common VM image that executes the cloud portion of our framework. We explain how applications are registered in the cloud using our client tools, instantiated on-demand by Generic Workers, and then invoked by clients using our client tools or APIs. We also discuss how our framework provides transparent and efficient file movement between client, cloud persistent store, and cloud VM local store.

3.1. **Application Deployment**

Applications need to be registered with the Generic Worker Registry before they are available for execution in the cloud. This involves a two-step deployment process. The first step, *Application Upload,* copies the application binaries and dependency libraries into cloud storage. The second step, *Application Registration*, informs the Generic Workers of the application location in cloud storage and its parameter signature. Our *register.exe* client tool performs both these operations and we persist registration information in Azure tables.

In the application upload step, users provide a desktop directory location that has all the binaries and dependency libraries required to be present in the VM for execution. This self-contained directory is packaged into a compressed file and uploaded into Azure’s persistent blob storage by our *register* tool. Outside any input parameters, the contents of that directory and the OS image will be the only set of files the Generic

```
(a) class SampleCloudApp.MathOps {
     int Add(int i, int j) { return i + j; }
     int Mult(int i, int j) { return i * j; }
   }
(b) > register -type .net -name MyMathOps
       -class SampleCloudApp.MathOps
       -appDir c:\SampleCloudApp\bin
   > register -type bin -name MyBlastAll
       -cmd “blastall –p blastn –d refseq_rna
   -i {1} -o {2}”
       -in #1 file  #2 string -out #2 file
       -appDir c:\ncbi\blast\bin
```

Figure 1: Registering applications with Generic Worker.framework using *register.exe* tool (a) Sample *MathOps* .NET class whose *Add* and *Mult* methods will be register as cloud applications. (b) *register* command to register *MathOps* class and *blastall* shell application.

Worker can access when executing the application. This sandboxing allows us to deploy an application on demand by just performing a directory copy from persistent cloud blob storage to transient VM local storage with no other state.

Users can specify a predefined application runtime when registering. This frees the user from managing the details of that runtime as well as eliminating the upload from desktop to cloud. Each predefined runtime is associated with invocation mechanics, environment configuration details, and a set of pre-packaged libraries or binaries that are available in Azure blob store. For example, a *Java1.6* runtime type would have the /java/bin directory present in the search path, set the JAVA_HOME environment variable and include JRE v1.6 files at that path. We support common eScience runtime types like .NET, Java and MatLab and allow more to be configured.

In the second step of application registration, the *register* tool stores details about the application in Azure table storage. These include a friendly name, runtime type, package location in blob store, parameter signature, and invocation target. The latter identifies the package entity to be executed. This may be a method in a class for .NET or Java applications, a compiled script for MatLab, or an executable file. In case of .NET or Java applications, users can register one or all methods in a class as applications and also automatically determine the input and output parameters using reflection. For other applications, the typed parameters are explicitly passed to the *register* tool commandline.

Figure 1 shows the *register* tool being used to register a .NET class and a commandline application. The first command in Fig. 1(b) uploads the DLLs for the *MathOps*.NET class in Fig 1(a) present in the .\SampleCloudApp\bin directory to Azure blob store and registers both methods, *Add* and *Mult*, with the Generic Worker Registry using the logical name of

```
(a) > invoke MyMathOps.Add 1 5
    Return value: 6
    > invoke MyBlastAll input.fasta output.txt
    Return value: c:\workdir-036\output.txt
    Download Console.Out file (y/n)?
(b) // int s = (new MathOps()).Add(1,5);
    Invoker invkr = new Invoker(“MyMathOps);
    int s = (int)invkr.Invoke(“Add",new[]{1,5});
```

Figure 2: Generic Worker commandline and API access to invoke applications. (a) *invoke.exe* shell tool used to call *MathOps.Add*method and *blastall* application in the Generic Worker VM (b) *Invoker* .NET utility used to run *MathOps.Add* application in the Generic Worker VM from a C# program.

MyMathOps. The inputs and output for both methods are determined by reflection. The second *register* command in Fig 1(b) deploys a *blastall* commandline application and passes the template shell command to be executed (-cmd “…”), the input and output parameter types and their mapping to the template. Corresponding steps in Figure 3 show the upload to blob store from desktop (R1) followed by registration of application (R2) and parameter signature (R3) with Azure tables.

3.2. Application Invocation

Applications written for the cloud are typically initiated by clients through custom web services in the cloud that launch the application in a VM using cloud queues for passing work requests monitored by the application from VMs, or by custom RPC mechanism shared by client and application. For applications and clients not designed for remote execution, each of these methods requires rewriting client and application code. Our command line option eliminates the need for such a rewrite. We also provide three programmatic ways – .NET API, REST service and workflow activity – to invoke a registered cloud application.

Our commandline tool, *invoke.exe*, takes the friendly name of the application and the list of input parameters. This tool internally uses our *Invoker* .NET client library, which .NET clients can also use directly. The *Invoker* API uses the passed application friendly name to query the Generic Worker Registry table for the application parameter details and then serializes the strongly typed input parameters into an XML message. File inputs are uploaded to blob store and their references passed, as discussed in Section 3.4.1 (Fig. 3, step 1). The *Invoker* also creates and passes the IDs of a cloud *App Response queue* for the application output and an optional log queue or cloud table for execution progress and provenance messages. The resulting XML work request message is placed in the common *Job Request queue* monitored by the Generic Workers (Fig. 3, step 2); the queue is accessed using the Azure REST web service API.

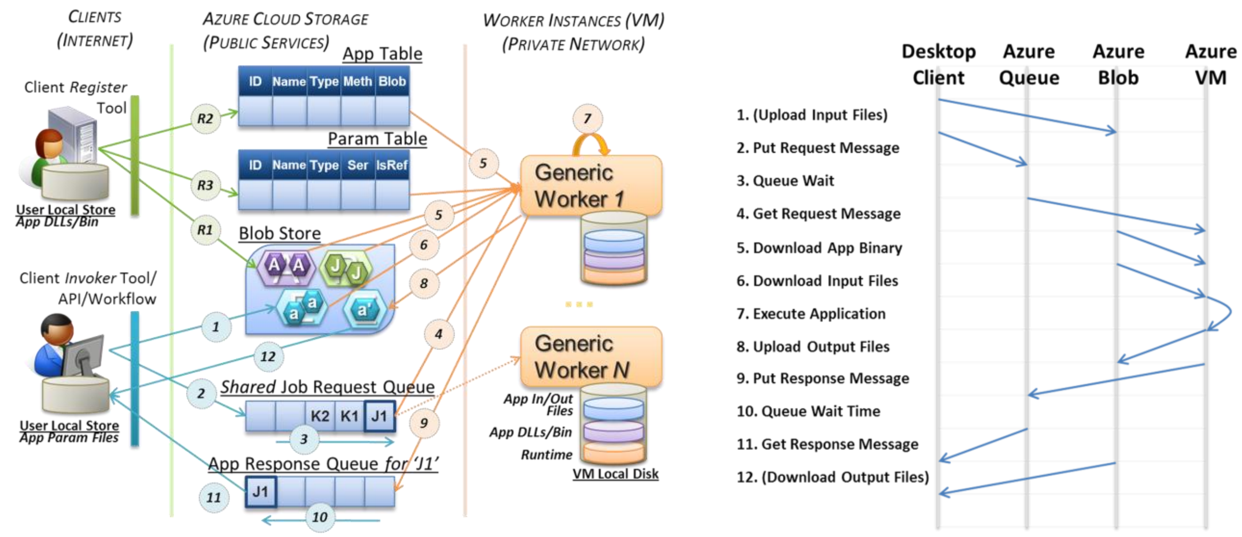

Figure 3. Generic Worker Architecture on Microsoft Azure (left) and Application execution sequence diagram (right). Numbered arrows R1-3 are application registration steps. Arrows 1-10 are application execution steps.

Our *Invoker* library monitors the newly created App Response queue for a matching response message to arrive (Fig. 3, step 11), deserializes the response (or exception) and returns (throws) it as objects to the client. In the case of the *invoker.exe* tool, this causes the response (or exception) to be printed on the screen. File references in the response are optionally dereferenced and downloaded from blob store (Fig. 3, step 12) as is the console output file. This automatic polling provides a more intuitive synchronous invocation for clients as opposed to the actual asynchronous model used through message passing.

Service oriented architecture clients or those written in non-CLR languages can directly call the Job Request queue's REST service to pass an appropriately formatted XML message and poll the App Response queue for the completion message. We also provide a *CloudApp* activity that allows cloud applications to be composed as part of workflows in the Trident Scientific Workbench [27].

3.3. Application Execution

The function of a Generic Worker is to execute any registered application on-demand while freeing users from having to manage the VM images, write roles or modify their application. When started, an Azure VM hands off control to the *Start* method of the only role deployed in the VM. In our case, this is the Generic Worker. Our worker starts in a clean OS VM and the only local files available are its own libraries.

A Generic Worker begins to monitor the common Job Request queue in the *Start* method. Workers process work requests in the order of arrival; since all workers are identical and do not persist local state in the VM, any worker can dequeue a work request and execute it. Users can start as many Workers as they wish to meet their application demand.

When a Generic Worker dequeues the next work request message (Fig 3, step 4), it checks the application name, verifies the parameters against the Registry tables, and, if necessary, prepares the VM for running with that particular application's runtime type (Fig. 3, step 5). As explained previously, for Java type applications, this involves copying the JRE files present in the Java package in Azure blob store to a local directory in the VM and setting the search and environment variables.

The Generic Worker then creates a working directory for this execution, downloads the application package from the Azure cloud blob store location listed in the Registry, and extracts it to the working directory (Fig. 3, step 5). Input parameters present in the work request are deserialized and dereferenced from blob store in case of files or large objects (Fig. 3 step 6). Next, the application is actually executed in the VM (Fig. 3, step 7). This may be through method reflection (.NET) or forking a separate process (Java, binary). Input parameters are passed either as objects (.NET) or in commandline (all others).

Once application execution completes, the Generic Worker collects results from the invocation. These are serialized back into strongly typed output parameters (or exception details in case of an execution error) as specified in the Registry entry for the application, and placed as an XML response message in the App Response queue specified in the work request (Fig. 3, step 9). The serialization includes persisting output files and console output to blob store with their references returned in the response message (Fig. 3, step 8). Finally, the Generic Worker cleans up the working directory, permanently deletes the original work request from the Job Request queue, and returns to polling the Job Request queue.

Because Generic Workers can run any registered application and they poll a common Job Request queue, the application is automatically load balanced across all active worker instances. Each Worker can also run multiple applications concurrently in separate threads, with a configured upper bound. Workers guarantee reliable completion of job requests by leveraging non-destructive read capability of Azure queues that revives messages when not permanently deleted within a certain period.

3.4. Data Access for Cloud Applications

In this section, we address the twin data access challenges of *transparent* and *efficient* file movement between desktop client, cloud persistent storage, and VM transient storage.

3.4.1. Transparent File Movement

Our Generic Worker and client tools enable desktop clients and cloud applications running in VMs to pass local files as parameters by transparently copying files between client local disk, VM local disk and cloud persistent store as necessary. This allows both clients and applications to access the files as if they were local, though created and used remotely.

When an application is registered, the parameters that correspond to input or output files are marked as a special *file* type in the Registry. These may be explicitly specified in case of commandline applications (e.g. for *blastall* in Fig. 1) or automatically determined for .NET applications when a *FileInfo* native type is used to pass file parameters. When our *Invoker* or a Generic Worker serializes input or output parameters, file parameters receive special handling. Rather than pass the entire file as input in the message (overflowing message size limits) or pass the local file path (which does not exist remotely), the file is transferred to the shared cloud blob storage and a reference to the blob location passed in the message. Upon deserialization, the blob file reference is downloaded from cloud storage by the *Invoker* or Generic Worker and a path to the local file passed to the desktop client or cloud application. That path is either a string for commandline applications or a *FileInfo* object for .NET types.

This has several advantages. First and foremost, eScience applications and clients can continue to use local files as parameters without modification. Files are transferred automatically using persistent, shared cloud storage as an intermediary. File path rewriting ensures that the correct local path is passed at all times. Lastly, very large files or memory objects that overflow message size limits can be chunked for transport.

3.4.2. Efficient File Movement

While transparent file transfer minimizes user complexity, transferring files between desktop and cloud storage can be costly in time and money. The download bottleneck can be due to bandwidth limits in the desktop client network or throttling in the cloud instance (~1-10MB/sec on Azure). Very large science data set transfers can also be costly monetarily ($0.10-$0.15/GB). Intermediate data in workflows can benefit from a more intelligent approach.

The cloud enables scientific workflows to run on a user desktop while the constituent activities run locally or in the cloud. When successive activities run as cloud applications, it is often unnecessary to download intermediate data produced by one activity to the desktop only to upload again for the next activity. Similarly, while it is often necessary to upload intermediate results from transient local VM storage to persistent cloud storage, it may be unnecessary to redownload the same data to the same VM prior to executing the next activity.

We provide a .NET *BlobFile* object type that abstracts whether the file is accessible via a local file path or cloud blob storage. Data transfer between local path and cloud storage occurs only when the application actually accesses the local path/blob location. For example, when a desktop client reads a *BlobFile*, the download occurs only if the client explicitly accesses the *LocalFile* field in the *BlobFile*. Similarly, no file transfer is necessary and only the blob storage location is passed when two cloud applications in a workflow pass *BlobFiles* objects. This can save both time and cost for data intensive applications. *CloudApp* activities in Trident Workflows use the *BlobFile* object by default for file parameters to optimize data transfers. .NET applications can also benefit from this feature by replacing the native *FileInfo* type with a *BlobFile* type.

Each Generic Worker also maintains a *file cache* within local VM disk storage. All VM-Cloud storage transfers pass through our write-through cache library that leverages the checksum/versioning feature of the Azure blob store. Prior to an application invocation, input blob parameters are downloaded from the Azure blob store only if there is a cache miss or the cache is dirty. After execution, dirty output blob parameters are uploaded from cache to the Azure blob store to prevent data loss. We use the cache not only for application data, but also any necessary application binaries and runtimes.

4. Evaluation for eScience Application

There are two main overheads in running applications using the Generic Worker. The first is a one-time overhead to start the required number of

Generic Worker VM instances to run applications at scale. This is amortized across all applications that run in the workers over time. The second is the application runtime overhead imposed by the framework. This includes time to upload and download input/output parameters and files to blob storage, the queue wait times, and time to download application binaries to VM. While the first two are common for any application running across desktop and cloud, the last is an overhead unique to the Generic Worker model.

VM Start Overhead. The Generic Worker imposes negligible VM start overhead. Figure 4(a) compares the time taken to start between 1 and 16 instances of the Generic Worker VMs against an equal number of blank VMs. In many cases, the Generic Worker VMs start faster than the blank VMs. This can be attributed to the inherent variability of VM startup times we have observed. For example, when starting 8 blank VMs, a difference of 3mins in startup times was seen between two runs as compared to the average of 5mins.

Runtime Overhead. The runtime overhead of the Generic Worker framework is measured for a Genome Wide Association Study (GWAS) .NET application with a Map-Reduce pattern [39]. Each Map process takes consumes a <1MB input file, runs for 10 mins, and creates an 8MB output file. The Reduce stage aggregates that output, also runs in about 10mins, and generates a single 8MB output file. The application has 120 libraries that total 45MB. The client to each stage runs on the desktop; the Map and Reduce run on Generic Workers in Azure.

Figure 4(b) shows the GWAS application with a Map fanout of 20 run on 20 Generic Workers. The primary overhead is the download of application binaries from blob storage to local VM store. This overhead of 17 seconds (half the total overhead) could be eliminated by our file cache for subsequent executions. The parameter upload and download times for the Map-Reduce calls are small (<2secs) since we use blob file references to reduce intermediate data transfer between desktop client and the workers. The queue wait times for the request and response messages are about 5 secs, which matches with the 5 secs polling delay in the client *invoker* library and Generic Worker. In a parallel application like GWAS, this has minimal impact on total application runtime. The time taken to run the actual GWAS application is 10mins and dwarfs the 35sec (~5%) overhead imposed by the Generic Worker.

Application Scalability. The Generic Worker model allows easy scale-out of applications. We measure the time taken to complete the GWAS application with a Map-Reduce fanout of 128 using 4 to 20 Generic Workers. We measure the total time taken to complete the application and compute the speedup with respect to running locally on an equivalent single core workstation that takes 129×10mins to finish (128 for map, 1 for reduce).

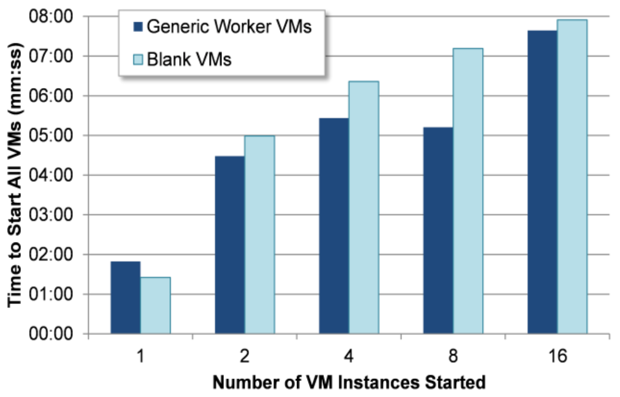

Figure 4(a). Average time in mins taken to start Blank Azure VMs vs. Generic Worker Azure VMs over 3 iterations.

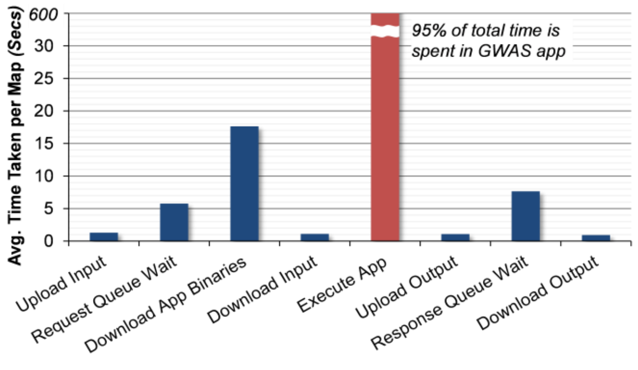

Figure 4(b). Average Overhead Time in secs contributed by each Generic Worker framework stage for GWAS application over 3 iterations. Total overhead is just 5% of the total GWAS runtime.

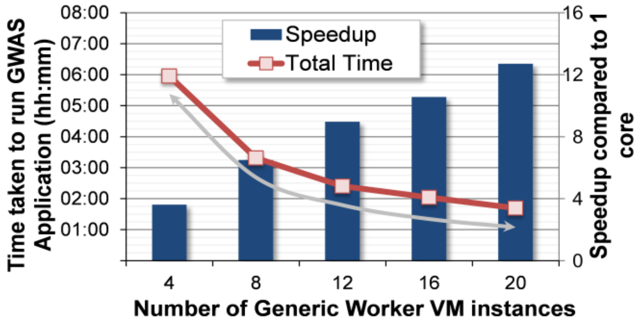

Figure 4(c). Total time (line graph) to complete GWAS application using increasing number of Generic Workers. Speedup (bar graph) as compared to a single core workstation that takes 21:30hrs. Reference line showing perfect linear speedup time shown in gray.

Figure 4(c) shows the total runtime and speedup of the GWAS application. The line graph shows that increasing the number of workers reduces the total time to complete GWAS linearly. While a perfect speedup when using, say, 12 VMs would be 12x compared to a desktop, we only observe a 9x speedup in the bar graph. The overheads discussed earlier cause this less than perfect speedup. However, even this speedup allows the Generic Worker framework scale out applications effectively.

5. Conclusion

In this paper, we have shown the benefits of using the Generic Worker model to easily deploy and execute applications in the cloud with minimal user effort. This further lowers the complexity barrier to entry for science applications. Our model imposes negligible overhead of 5% when scaling the GWAS application to 20 workers. Techniques such as on-demand data transfers and VM file caches can actually reduce the performance penalty for data transfer between desktop, cloud and VM storage. The linear speedup we observed shows the effectiveness of the Generic Worker to scale out applications to the cloud with ease.

As future work, we can include additional common eScience runtime types such as R. A similar Generic Worker model can also be considered for *nix IaaS platforms. While it is possible to support more complex deployment dependencies based on existing work, the goal of making the framework easily accessible should not be compromised.

Our stateless Generic Worker architecture allows dynamic load balancing to be easily implemented; while users can do this manually now, we envision adding more Worker VMs to add capacity for all queued applications and/or likewise for scaling down. Another scheduling optimization would leverage data locality of pipelined workflow applications by having Workers preferentially dequeue work requests whose input files are already present in their local file cache. Together, these lead us to an accessible science platform for the cloud that further enhances application scalability.

References

[1] Building Web Services for scientific Grid Applications, G. Kandaswamy, et al., *IBM J. Res. & Dev.* 50 (2/3) 2006.

[2] Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. S. Krishnan, et al., in *SERVICES* 2009.

[3] Above the Clouds: A Berkeley View of Cloud Computing, M. Armbrust, et al., *Technical Report* UCB/EECS-2009-28, UC Berkeley, 2009.

[4] The Eucalyptus Open-Source Cloud-Computing System, D. Nurmi, et al., in *CCGRID* 2009

[5] Cost-Benefit Analysis of Cloud Computing versus Desktop Grids, D. Kondo, et al., in *IPDPS* 2009.

[6] Resource Monitoring and Management with OVIS to Enable HPC in Cloud Computing Environments, J. Brandt, et al, in *IPDPS* 2009.

[7] Cloud Computing for the Sciences, F. Sullivan, *CiSE* 11(10) 2009.

[8] A Break in the Clouds: Towards a Cloud Definition. L. M. Vaquero, et al., *Comp. Comm. Rev.* 39(1), 2008.

[9] Slow Moving Clouds Fast Enough for HPC, M. Feldman, *HPC Wire*, August 10 2009.

[10] CloudBurst: Highly Sensitive Read Mapping with MapReduce, M. Schatz, *BioInformatics* 25(11), 2009

[11] Cloud Computing and Grid Computing 360-Degree Compare, I. Foster, et al, in *GCE* 2008.

[12] Cloud Computing for e-Science with CARMEN, P. Watson, et al, in *IBERGRID* 2008.

[13] Commodity Grid Computing with Amazon's S3 and EC2, S. Garfinkel, *USENIX Magazine*, 2007.

[14] Emergence of the Academic Computing Clouds. K. A. Delic, and M. A. Walker, *Ubiquity* 2008.

[15] GADe: Toward Automatic Deployment of Applications on Computational Grids, S. Lacour, et al. in *Grid* 2005.

[16] Coming Soon: Research in a Cloud, P. F. Gorder, *CiSE* 10(6), 2008.

[17] Virtual Infrastructure Management in Private and Hybrid Clouds, B. Sotomayor, et al., *Internet Comp.*, 13(5) 2009.

[18] Approaches for Service Deployment, V Talwar, et al., *Internet Comp.*, 9(2), 2005.

[19] Distributed Ant: A System to Support Application Deployment in the Grid, W. Goscinski, et al., in *Grid*, 2004.

[20] Cloud Computing for Small Research Groups in Computational Science and Engineering: Current Status and Outlook, H. Truong, et al., *Technical Report*, Vienna Univ. of Tech., August 2009.

[21] Cloud Computing – Issues, Research and Implementations, M. A. Vouk, in *Info. Tech. Interf.*, 2008.

[22] http://www.microsoft.com/windowsazure/

[23] http://www.rackspace.com/

[24] http://developer.amazonwebservices.com

[25] Management Framework for Amazon EC2, F. Bitzer, *Technical Report* No. 2841, University of Stuttgart, 2009.

[26] Amazon S3 for Science Grids: a Viable Solution?, M. Palankar and A. Iamnitchi, in *DADC* 2008

[27] Building the Trident Scientific Workflow Workbench for Data Management in the Cloud, Y. Simmhan, et al., in ADVCOMP 2009.

[28] ADEM: An Automation Tool for Application Software Deployment and Management on OSG, Z. Hou, et al., in *Grid* 2009

[29] Scientific Computing in the Cloud, J. Rehr, et al., *CiSE*, preprint, Jan. 2010.

[30] Sky Computing, K. Keahey, et al., *Internet Comp.*, 13(5), 2009.

[31] Cloud Computing and the Common Man, J. Viega, *Computer*, 42(8), 2009

[32] Real-time Streaming of Environmental Field Data, E. R. Vivoni, et al., *Computers & Geosciences*, 29(4), 2003.

[33] Cloud Computing for Science, K. Keahey, in *SSDBM* 2009.

[34] Pegasus: Mapping Large-Scale Workflows to Distributed Resources, E. Deelman and G. Mehta, in *Workflows in e-Science*, Springer, 2006.

[35] A Profile-Based Approach to Just-in-Time Scalability for Cloud Applications, J. Yang, et al. in *IEEE Cloud*, 2009.

[36] Scientific Workflows as Services in caGrid: A Taverna and gRAVI Approach, W. Tan, et al., in *ICWS* 2009.

[37] Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, M. Isard, et al., in *EuroSys* 2007.

[38] http://hadoop.apache.org

[39] Improving Detection in Large-Scale Genetic Association Studies, Jennifer Listgarten in *Microsoft Faculty Summit*, 2009.