CloudScale – a Novel Middleware for Building Transparently Scaling Cloud Applications

Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, 1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

With the promise of seemingly unlimited IT resources, the trend of cloud computing is currently revolutionizing software engineering. However, at the moment, building applications for the cloud is a rather cumbersome and manual task. In this paper, we introduce the CloudScale middleware for building applications on top of Infrastructure-as-a-Service (IaaS) cloud offerings. CloudScale allows developers to build cloud applications like regular Java programs, without dealing with the intricacies of cloud hosts (virtual machine) management, remoting, and code distribution, without handing off control over the physical distribution of their application to commercial Platform-as-as-Service (PaaS) providers. We numerically evaluate the overhead introduced by CloudScale based on an example application, and discuss advantages and limitations of the system as compared to manually deploying the application on an IaaS cloud.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming; H.3.5 [**Online Information Services**]: Web-based Services

General Terms

Management, Performance, Measurement

Keywords

Cloud Computing, Middleware, Scalability, Programming Models

1. INTRODUCTION

In the last years, cloud computing [2], most importantly the Infrastructure-as-a-Service (IaaS) paradigm [5, 8], has been established as a global trend towards more "elastic" provisioning of IT resources. Resources, such as computing power or data storage, are no longer provisioned only on-premise based on predictions of worst-case scenarios, but instead flexibly rented on demand for just as long as the resources are actually needed. Cloud computing usually comes with pay-as-you-go pricing policies, which allow to view IT costs as expenses rather than investments, that is, the price tag of cloud computing is generally strictly usage-based without any explicit upfront investments. This policy is closely related to the advantage most commonly associated with cloud computing over traditional on-premise IT: in cloud computing, IT costs can be kept low by reducing the upfront infrastructure investments close to zero, and paying only what is actually used. In this way, IT costs can be viewed as an expense rather than an investment [2]. Furthermore, the flexibility offered by cloud computing enables novel business models for start-up companies, which would have been too risky a few years ago. As Amazon's Jinesh Varia puts it: "In the past, if you got famous and your systems or your infrastructure did not scale you became a victim of your own success. Conversely, if you invested heavily and did not get famous, you became a victim of your failure." [18].

However, at the moment, it is not easy for application developers to realize the advantages promised by the cloud. IaaS, despite all its features, provides a very low-level abstraction (virtual machines), leaving the intricate details of building scaling cloud applications to the developer. Platform-as-a-Service (PaaS) offerings, such as Windows Azure[1] or Google Appengine[2], offer more developer support, but come with their own limitations, including a tight vendor lock-in, and limited control over the scaling and deployment behavior of applications.

This paper introduces CloudScale, a Java-based middleware for building abstract cloud applications, which seem like regular Java applications but can easily be deployed onto common IaaS clouds. CloudScale implements a declarative deployment model, in which application developers specify the scaling requirements and policies of their applications as code annotations. To the developer, the application seems like a local Java application. All distribution and scaling related code is introduced at application startup time, using aspect-oriented programming [12] (AOP) and bytecode manipulation. This relieves the developer from dealing with intricate cloud deployment issues, without surrendering them to a commercial third party, such as Google or Microsoft.

[1]http://www.microsoft.com/windowsazure/
[2]http://code.google.com/appengine/

SAC'12 March 25-29, 2012, Trento, Italy.

Note that, even though we present a concrete Java-based middleware in this paper, a similar middleware can easily be implemented, e.g., on top of C#/.NET. Hence, the general approach presented in this paper is not limited to Java, even though we base our discussion on Java tooling.

The rest of this paper is structured as follows. Section 2 illustrates the positioning of the CloudScale middleware, and introduces a case study application. Section 3 presents the actual contribution of the paper, the CloudScale middleware. This middleware is evaluated in Section 4. Some scientific related work is presented in Section 5. Finally, the paper is concluded in Section 6.

2. MOTIVATION

In the following, we illustrate the ideas behind the development of CloudScale based on an existing computing-intense scientific prototype. Concretely, we use the cost-based optimizer component from the PREvent toolkit, as discussed in [7]. The cost-based optimizer uses various optimization algorithms, for instance memetic algorithms [14], to identify the cost-optimal way to modify a running Web service composition [4]. This tool is a non-trivial Java-based application consisting of about 8000 source lines of code, excluding empty lines and generated code. An earlier version of the tool has been distributed as part of the VRESCo project [10] and is available online[3]. However, the concrete goals and implementation of this tool are not important for this paper.

As optimization is computationally expensive, there is evidently a strong incentive to deploy this optimizer tool into an IaaS cloud, so that more optimization instances can be handled in parallel. Using state-of-the-art technology, this deployment is not trivial. One needs to split the application into task manager and workers, setup virtual machines, install the respective application parts on the virtual machines and, at runtime, monitor the virtual machines to make sure that the application is not over- or underprovisioned. If the application is deployed to the AWS cloud[4], tools such as AWS Elastic Beanstalk (deployment) or Amazon CloudWatch (monitoring) can be used to ease these tasks to some extend. However, even using these advanced tools, cloud deployment quickly becomes a labor-intensive chore. If the optimizer tool should be deployed on a private cloud, for instance based on Eucalyptus[5], the developer is mostly on her own, anyway.

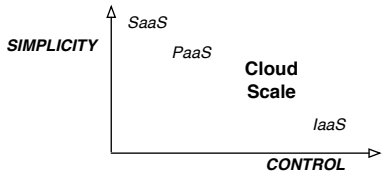

Figure 1: Taxonomy of Cloud Computing Offerings

With the CloudScale middleware, we aim at easing deployment scenarios, such as this one. Our goal is that developers can deploy applications to any IaaS cloud with very limited changes to the application source code, and without explicitly setting up and managing virtual machines. In the larger taxonomy of Cloud Computing offerings [5], CloudScale fills the space between PaaS and IaaS (Figure 1). It offers more control over the behavior of the application than typical PaaS solutions, and demands less programming, deployment and monitoring effort than IaaS solutions. However, it should be noted that CloudScale is used on top of IaaS, i.e., "under the hood" the application is still executed on a regular IaaS offering, such as AWS EC2 or Eucalyptus.

[3]http://sourceforge.net/projects/vresco/files/
[4]http://aws.amazon.com/
[5]http://www.eucalyptus.com/

3. CLOUDSCALE FRAMEWORK

On a high level, the CloudScale middleware handles two tasks for applications. On the one hand, the middleware manages virtual computing resources in the cloud, referred to as cloud hosts (*CHs*). This includes monitoring the load of *CHs*, instantiating new *CHs* if necessary (scaling up) or terminating existing ones (scaling down). Furthermore, this task includes cost control, i.e., making sure that the cloud setup does not violate user-specified cost constraints. On the other hand, the framework is responsible for managing cloud objects (*COs*). *COs* are regular program-level objects, which are abstractions of application logics, which should be distributed over a cloud. Optimally, *COs* are highly cohesive and very loosely coupled to the rest of the application. In the optimizer tool, good *COs* are the classes implementing the optimization: they do not depend much on the rest of the application, but their execution is very computing-intense (i.e., it makes sense to distribute those objects). For each *CO*, the CloudScale middleware decides if the object should execute locally, or alternatively selects a suitable *CH* to deploy it on. *CHs* host *COs*. Any given *CH* can host one or more *COs*. These *COs* are not restricted to having the same type, i.e., every *CH* can host *COs* of various different types.

```
1  MemeticAlgorithm.java:
2    @CloudObject(cloudConfig = "config.props",
3      scaling = RoundRobinScaling.class)
4    public class MemeticAlgorithm { ... }
5
6  ClientApp.java:
7    MemeticAlgorithm ma = new MemeticAlgorithm();
8    ma.init(...);
9    ma.solve(...);
10   ma.getResult(...);
```

Figure 2: Example Client Application

To the client application, both tasks are transparent, i.e., the application does not need to know which *CHs* exist, or which of its *COs* is executed on which *CH*. In fact, to the client application, CloudScale is almost invisible. This is indicated in the code snippet in Figure 2. The class `MemeticAlgorithm` is declared as a *CO*, and a round robin strategy is used for distributing *COs* over *CHs*. The usage of this class (in *ClientApp.java*) is in no way special.

The UML Collaboration Diagram in Figure 3 exemplifies this based on Line 9 in Figure 2. The dashed arrow is what the interaction seems like to the client application. The filled arrows are what is actually executed by CloudScale: the *CO* is transparently replaced by a client proxy object, which notifies the local cloud manager about the planned invocation. The cloud manager looks up the *CH* hosting the actual object, and forwards the invocation to the CloudScale

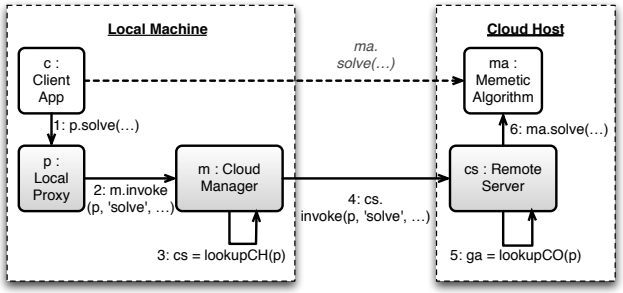

Figure 3: Example CloudScale Interaction

server component running on this host. Afterwards, the server finds the correct object from its internal cache, and finally issues the actual invocation. This implementation is essentially an instantiation of the `Broker` remoting pattern, as discussed in [20].

3.1 Middleware Overview

In the following, the basic architecture of the CloudScale middleware is described. Figure 4 depicts the most important components of a typical CloudScale application. Essentially, the middleware consists of a client library, which is deployed along with the client application. Additionally, the framework utilizes an arbitray number of *CHs*, which run a specific CloudScale server component. *CHs* are responsible for executing *COs*. There is a fixed pool of static *CHs*, which is co-used by an arbitrary large number of client applications. In addition, every application can scale up by instantiating new on demand *CHs*. These are identical to the hosts in the static cloud pool in every respect, except that they are terminated as soon as the client application releases them. The component responsible for deciding if new on demand *CHs* are needed, or if existing ones should be terminated, is the scaling manager. This is done by interpreting the scaling policy that is given by the client application as part of the *CO* definition (see Listing 2, Line 3). Obviously, scaling policies are usually based on monitoring information, as delivered by the host monitor. The CloudScale approach to monitoring and scaling is described in more detail in Section 3.2.

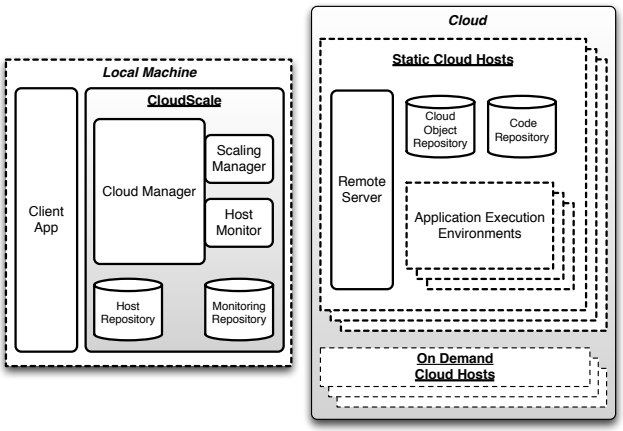

Figure 4: Architectural Overview

Both, static and on demand *CHs*, contain a remote server component as central element. The remote server is responsible for communicating with the cloud manager on the client side. Furthermore, *CHs* contain a *CO* repository, which stores all *COs* which are not currently executing. As soon as an invocation for such an object is received by the remote server, it will load the object from the repository, move it into an application execution environment, and invoke the requested method. Application execution environments shield *COs* from each other, i.e., they make sure that CloudScale hosts are multi-tenancy enabled. Additionally, *CHs* contain a code repository, which stores the bytecode of *COs*. This code is transferred to the host when a new class of *CO* is deployed for the first time. The exact process of code distribution is detailled in Section 3.3.

3.2 Cloud Object Instantiation and Scaling

One central task of the middleware is the management of *CO* creation and deployment. Essentially, whenever the client application creates a new instance of a *CO*, the CloudScale cloud manager component has to execute the following steps. Firstly, the newly created local object is replaced with a proxy, redirecting all future method invocations to the cloud manager. Secondly, the cloud manager queries the scaling manager component for a suitable *CH* to deploy the new *CO* to. Essentially, the scaling manager has three possible options: (1) use an existing *CH* (e.g., because there are sufficient resources to host the object), (2) instantiate a new *CH* and use it, or (3) deem that, based on current policies, no *CH* will be able to execute the *CO*. Thirdly, if a suitable *CH* has been found or created, the cloud manager contacts the server component on the selected *CH* and requests it to deploy the *CO*.

From these three steps, the selection of a *CH* is the most interesting. This step is based on scaling policies, which can be declaratively defined either for a single *CO* (as done in Figure 2) or globally for all *COs* of an application. Evidently, it is not feasible to define a single policy, which behaves as expected for each application that can be built using CloudScale. Hence, the middleware only provides a limited set of simple default policies, such as `Round-Robin-Scheduling` (assign *COs* in turn to each static *CH*, without instantiating on demand *CHs*) or `One-CO-Per-CH` (assign at most one *CO* to each *CH*, and instantiate new *CHs* if no free *CHs* are available anymore). Typically, a client application will define or derive its own custom policies, which are developed as regular Java classes implementing a well-defined interface.

Scaling policies are usually defined on top of monitoring information, as collected by the host monitor component. At the moment, the host monitor periodically collects for each *CH* (1) the average CPU utilization in the monitoring period, (2) the average RAM utilization, (3) the average free disk space, and (4) the data transfer into and out of each *CH*. Based on this low-level load data, useful scaling policies can be defined. Specifically, in conjunction with pricing information of the IaaS provider, this data can be used to monitor the costs of the cloud deployment, and implement scaling policies based on costs.

Figure 5 depicts the lifecycle of a *CO* in CloudScale. Firstly, when an object is created, it immediately goes into the **Idle** state. In this state, the object is fully functional and awaits invocations. If a method is invoked, the object goes into

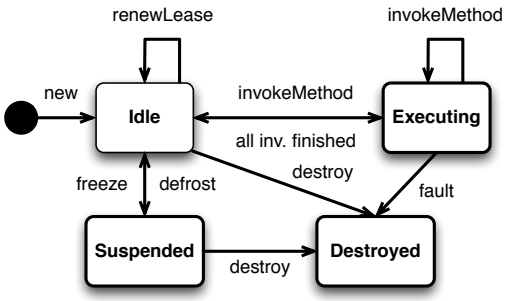

Figure 5: Cloud Object Lifecycle

the **Executing** state. Note that additional invocations can asynchronously be executed on a single *CO*, just like on a regular Java object. When all method invocations are finished, the *CO* goes back into the **Idle** state. If no invocations for a *CO* are received for a defined period of time, the object is moved out of the main memory and saved to a permanent storage (either a disk or database), if possible. The *CO* is then said to be in the **Suspended** state, and is loaded back into memory when the next invocation for it is received. If it is not possible to suspend an object to a permanent storage (i.e., if the object or its state cannot be serialized), the object moves directly into the **Destroyed** state and is garbage collected. If a client application wishes to prevent its *COs* from being suspended or destroyed, it can issue requests to renew the lease on the *COs*. Finally, objects can be destroyed from each state, either by explicitly requesting destruction (by the client application) or by an unrecoverable fault during the execution of a method invocation. Note that a client cannot request object destruction while the *CO* is still in the **Executing** state, i.e., while an invocation is still ongoing.

3.3 Code Distribution

In order to be able to execute *COs* remotely, some means to transparently distribute application code between client application and *CHs* is necessary. This is similar to the idea of mobile code [3], as often discussed in the context of agent-based systems [6]. Essentially, as part of *CO* deployment, the cloud manager component needs to make sure that the *CH* has the correct version of the application code available, and that all dependencies of the application are met. At the moment, we assume that both, the client application and all its dependencies, are available as Java archives, so that it is easy to physically transport code over the network.

```
1  // In: Cloud Object co, Remote Server cs
2  // Out: void
3  transportCode(co)
4    appCode = getExecutingJar(co)
5    hash = crc(appCode)
6    if(!cs.isInCache(hash))
7      uploadCode(app_code)
8
9    dependencies = getDependencies(co, STRATEGY)
10   foreach(dependency in dependencies)
11     hash = crc(dependency)
12     if(!cs.isInCache(hash))
13       uploadCode(dependency)
```

Figure 6: Code Distribution Procedure

The overall procedure of code distribution is sketched in Java-like pseudocode in Figure 6. Firstly, the Java archive containing the *CO* is identified, and a cyclic redundancy checking (CRC) hash is generated. Using this hash, we can check whether the remote server component running on the *CH* already has a copy of this archive in its internal code repository. If this is not the case, we upload the archive. In a second important step, we now need to identify which additional archives this application requires. For each of those archives, we proceed in the same way as for the application archive itself.

CloudScale supports three different strategies for tracking the dependencies of an application. The most simple strategy is to assume that the entire classpath (excluding system libraries) is required by the *CO* (classpath strategy). This strategy is simple, relatively efficient to implement, and requires no further configuration by the developer. However, in many cases this pessimistic strategy will upload plenty of code that is actually not needed, increasing the overhead of the CloudScale framework and taking up unnecessary disk space on the *CH*. Another conceptually simple strategy is to let the developer explicitly specify the dependencies of each *CO* (explicit strategy). This strategy is also very efficient to implement, and (assuming that the developer specifies only what is actually needed) reduces the overhead introduced by code distribution to the minimum. However, for the developer, this strategy might be cumbersome, as she has to track dependencies for each *CO* manually. The final dependency tracking strategy is to analyze the `import` statements of the *CO* and each class used by the *CO* (recursively), and automatically identify in this way, which Java archives are actually used (import strategy). This strategy also reduces the runtime overhead of code distribution to the minimum, and does not require any further configuration. However, for complex objects this tracking is computationally expensive. Furthermore, runtime classpath manipulation or class loading will derail this strategy, leading to execution time errors. The concrete strategy to use can be specified declaratively by the application. By default, the classpath strategy is used.

3.4 Implementation

Technically, the CloudScale middleware has been implemented using aspect-oriented programming techniques, as provided by the AspectJ[6] framework. AspectJ enables us to shield the user almost entirely from the middleware, as all necessary code changes (e.g., replacing application objects with proxies, brokering invocations via CloudScale, inserting and executing the necessary code for *CH* management) can be done at load-time of the JVM (load-time weaving).

Minimally, the only four simple steps required for an application developer to start using CloudScale are to annotate the objects to distribute, as in Figure 2 with `CloudObject` annotations, provide a configuration file that contains some necessary cloud configuration parameters (e.g., access credentials for starting and terminating on demand *CHs*), add a command-line parameter to enable load-time weaving in the JVM (`-javaagent:lib/aspectjweaver.jar`), and provide an `aop.xml` configuration to enable the single aspect utilized by CloudScale. The most primitive version of this configuration file is provided in Figure 7, but additional AspectJ directives can of course be added.

[6]http://www.eclipse.org/aspectj/

```
<aspectj>
  <aspects>
    <aspect
      name="org.acm.sac12.CloudObjectAspect"
    />
  </aspects>
</aspectj>
```

Figure 7: Enabling CloudScale in an Application

Hence, it is extremely easy to enable and disable CloudScale for an application, as one only needs to omit the respective command-line parameter or disable weaving for the `CloudObjectAspect`, and the client application will execute locally, without being touched by CloudScale at all.

The `CloudObject` annotation, which is the only mandatory code-level change required for using CloudScale, takes three optional parameters. Firstly, using the `cloudConfig` parameter, a developer can specify the location of the CloudScale configuration file (if no custom location is given, the configuration is expected to reside in `META-INF/cloudconfig.props`). Secondly, the scaling policy to use for this *CO* can be specified using the `policy` parameter. Thirdly, developers can specify the dependency tracking strategy for this *CO* using the `dependencyStrategy` parameter. In addition, developers may annotate methods of *COs* using the `DestructCloudObject` annotation. This tells the middleware that the *CO* is to be moved into the **Destroyed** state after the invocation of this method has finished successfully.

To ease integration with external tooling, the CloudScale remote server component (see Figure 4) is implemented as a RESTful Web service [15] using the Apache CXF[7] framework. This allows us to query the status metadata provided by *CHs* (e.g., which objects are currently hosted, what is the state of a given *CO*, what is the current load of the host?) using standard tooling, e.g., Web browsers. The code repository currently uses a simple file system based implementation. Conversely, the cloud object repository is implemented as in-memory database. For the Eucalyptus IaaS cloud, an Ubuntu 10.04 Lucid Lynx based EMI (Eucalyptus Machine Image), which launches all these server-side components on startup, is included as part of the CloudScale middleware. In order to use it with other clouds, e.g., Amazon EC2, a custom image that starts these components needs to be built. We plan to release the first version of CloudScale as open source software in the near future.

4. EVALUATION

In order to evaluate the CloudScale middleware, we will firstly discuss some numerical observations based on real usage of CloudScale for scaling a non-trivial application (the optimization toolkit described in Section 2). Afterwards, we qualitatively discuss the implications of those numerical results, and the advantages and disadvantages of the current incarnation of CloudScale.

4.1 Numerical Evaluation

In this section, we discuss numerical results measured from using CloudScale to deploy a real application on a real IaaS cloud. As IaaS infrastructure, we used a private cloud implemented using Eucalyptus. The cloud setup consisted of a cloud controller and four node controllers, each running on a dedicated Dell blade server with two Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) and 16 GByte RAM. We implemented three different cases. Firstly, we started the optimizer on a regular desktop notebook (Macbook Pro 2.7 GHz Intel Core i7) and measured the performance in this 'non-cloud' setup. This reflects the way how the optimizer has been used so far. Secondly, we manually deployed the optimizer on the Eucalyptus cloud described above. Thirdly, we used CloudScale to deploy the application to said cloud. Then we fed requests at an increasing rate to the different setups, and measure the median time that the tool took to process a single request. Note that for both types of cloud deployment, a separate cloud machine was dedicated to each request (i.e., for 10 parallel requests, we utilized 10 cloud virtual machines). All experiments are the average of 30 repetitions, to reduce natural variations in the duration of optimization. The results of this experiment are shown in Figure 8. On the x-axis, the number of parallel requests are plotted. The y-axis denotes the median time in seconds for processing one request, i.e., for completing one optimization.

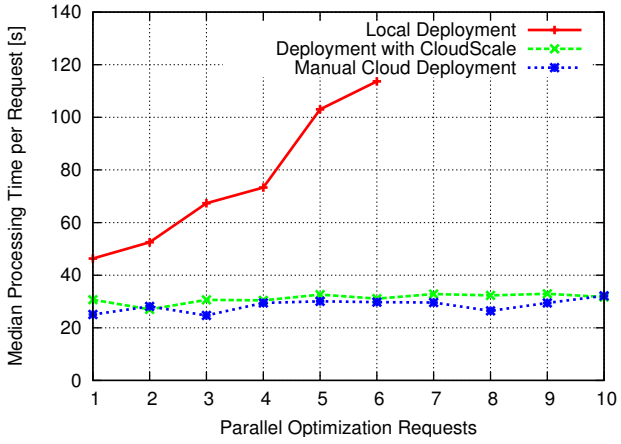

Figure 8: Numerical Evaluation Results

In the figure, we can see that the 'non-cloud' setup of the tool does, in fact, not scale at all. Optimization is computationally expensive, and the median time for processing a single request increases linearly with the number of requests that have to be handled in parallel. Hence, we have stopped this experiment at six parallel requests. Both cloud setups fare much better than that. For 10 parallel requests, which already constitute full load on 10 virtual machines, we cannot see a significant increase in the median processing time. Realistically, we can assume that the median processing time is increasing slightly, but this factor is small as compared to the total time necessary for optimization. Furthermore, we assume that CloudScale will, in the long run, be slightly slower than the manually distributed setup, as the additional indirection of CloudScale obviously induces some overhead. However, in this experiment, this overhead is small enough not to be statistically significant.

Another performance-related concern of using CloudScale is that load-time AspectJ weaving potentially increases the startup time of the application. In the case of the optimizer tool, we experienced only a very minor startup time degration (from three to four seconds), which was irrelevant to us.

[7]http://cxf.apache.org/

However, if the overhead of load-time weaving is problematic, it is also possible to weave the necessary code changes statically, as part of the application build process.

4.2 Discussion

As the numbers presented in Section 4.1 show, parallelizing the PREvent optimizer tool in the cloud makes a lot of sense. It can be argued, that similar performance gains can also be reached for other computing-intense applications, e.g., applications from the ara of scientific computing. Furthermore, we can see that using the CloudScale middleware induces a very small overhead over completely manual distribution in this case. For other applications, especially those where single requests are faster served (e.g., typical Web applications, where load is typically produced by the concurrent execution of a very large number of simple tasks, such as serving an HTML page), developers need to assess themselves whether the slight performance hit of using another layer of indirection on top of the IaaS cloud is outweighted by the advantages of the simpler programming model proposed by CloudScale. As part of our future work, we plan to conduct further research with regard to the suitability of CloudScale for Web application development.

It should be noted, that there are a number of technical and conceptual limitations in the current incarnation of CloudScale. Firstly, all parameters of constructors and non-private methods of *COs* need to be marked as `Serializable`, as the middleware needs to be able to transport parameters to the executing *CH*. The same holds true for return values. Secondly, at the moment, all non-private methods and constructors in *COs* use call-by-value semantics, which the developer needs to be aware of. Essentially, this means that if a developer passes the same object as parameter to the invocation of two different *COs*, the invocation will be executed on two different copies of the object. We plan to add call-by-reference semantics in a future version of the middleware. Thirdly, we currently do not support static fields in *COs*, i.e., value changes in static fields are not propagated to other instances of the object running on different hosts. This feature is also scheduled for a future version of the middleware. However, all of these limitations are to some extend also prevalent if the application developer chooses to cloud-deploy the application manually. That is, all these limitations are actually more characteristics of distributed computing, and not specific to the CloudScale middleware.

5. RELATED WORK

Various research efforts have previously tackled the problem of providing scalable software platforms in clouds. We discuss the ones we consider most important in the following. Most existing work has focused on efficiently deploying applications in the cloud. For instance, Mietzner et al. [11] present the composite cloud application framework *Cafe*. Liu [9] draws an analogy between virtual machine images/instances and Java classes/objects, respectively, and introduces the Rapid Application Configurator (RAC), which seamlessly integrates with Amazon EC2.

Foundational work on distributed deployment and hosting of (Java) applications dates back to the era before Cloud computing. Paal et al. [13] discuss important issues related to class loading and namespace separation, which are also relevant to our work. Our approach is also related to replicating and clustering of Enterprise Java Beans (EJB), which has been evaluated in [21]. However, most EJB implementations lack the dynamism of CloudScale, and do not provide resource management strategies or transparent code distribution. More recently, Sampaio et al. [16] have claimed that programmers would ideally use a "single shared global memory space (heap of objects) of mostly unbounded capacity". In fact, CloudScale attempts to achieve a similar goal by providing developers (near-)infinite resources while maintaining the view of a program running locally.

Moreover, specialized programming models have been proposed to let developers take direct advantage of the cloud computing paradigm. *Aneka* [19] is a platform for deployment of .NET-based applications, employing a specialized programming model. The *BOOM* initiative at UC Berkeley aims at simplifying declarative programming for the cloud [1]. Whereas BOOM is mostly suited for data analytics, CloudScale is targeted at general-purpose programming and deployment of arbitrary Java applications. *Ibis* [17] is a Java-based programming environment tailored to parallel computations in Grid environments. At its core, the platform defines the Ibis Portability Layer (IPL), a set of communication primitives akin to the Message Passing Interface[8] (MPI).

PaaS is a lucrative market and a plethora of commercial offerings have recently sprung into existence (e.g., *CloudBees*[9], which provides continuous integration of Java applications). From a technical viewpoint, these providers do little more than allowing scalable remote access to existing software developing platforms, enriched with rich user interface and Web 2.0 experience. The *Salesforce*[10] platform goes beyond this point and defines the tailor-made *Apex*[11] programming language and proprietary APIs for access to cloud storage. The main difference between CloudScale and those commercial SaaS and PaaS solutions is that, when using CloudScale, developers retain full control over their application. That is, even though CloudScale hides some scalability-related issues from developers, they are still free to customize the way CloudScale works to their own needs, either by implementing custom scaling policies, adapting the CloudScale framework itself, or managing some *COs* in the application manually.

The implementation of the CloudScale middleware is conceptually similar to existing state-of-the-art tooling for Java-based remoting, e.g., Java RMI, Enterprise Java Beans (EJB) or CORBA. These frameworks, while technically similar, provide only limited support for automated scaling. However, the ideas behind the CloudScale middleware could be implemented on top of these technologies, enhancing, e.g., the CORBA remoting model with automated load balancing in the cloud.

6. CONCLUSIONS

In this paper, we have introduced the CloudScale middleware for transparently scaling applications using an IaaS cloud. The CloudScale approach provides a middle ground between common IaaS offerings, which provide great control over the application, but do so at the costs of high deployment effort, and PaaS offerings, which are easy to use, but

[8] http://www.mpi-forum.org/docs
[9] http://cloudbees.com
[10] http://salesforce.com
[11] http://www.salesforce.com/us/developer/docs/apexcode/

provide little control. The only change on source code level necessary for using CloudScale is the introduction of some declarative statements, to designate which application objects should be distributed over the cloud. CloudScale relieves the developer from creating and managing virtual machines in the cloud, and automatically monitors the utilization levels of used resources, scaling up and down as necessary. We have discussed some performance aspects of using the CloudScale middleware based on an existing Java-based application.

As next steps, we plan to extend the middleware in the following directions. Firstly, in the current version of CloudScale, *COs* are allocated to *CHs* when they are created, based on the system state at creation time. In the future, we will provide means to migrate running *COs* from one *CH* to another, if the system state has significantly changed. Secondly, we will extend the framework with support for other types resources besides computing, e.g., data storage services, such as Amazon S3. Thirdly, we will provide means for monitoring *COs* beyond generic performance data, such as CPU load or RAM and disk utilization. To this end, the next version of CloudScale will provide an interface to plug in application-specific performance metrics (e.g., solutions evaluated per second, in the case of the cost-based optimizer tool). These application-specific metrics can then be used in custom scaling policies.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreements 215483 (S-Cube) and 257483 (Indenica).

7. REFERENCES

[1] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *5th European Conference on Computer Systems (EuroSys'10)*, pages 223–236. ACM, 2010.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32, New York, NY, USA, 1997. ACM.

[4] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[5] D. Hilley. Cloud Computing: A Taxonomy of Platform and Infrastructure-Level Offerings, 2009.

[6] N. R. Jennings. An Agent-Based Approach for Building Complex Software Systems. *Communications of the ACM*, 44:35–41, April 2001.

[7] P. Leitner, W. Hummer, and S. Dustdar. Cost-Based Optimization of Service Compositions. *IEEE Transactions on Services Computing (TSC)*, 2012. To appear.

[8] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's Inside the Cloud? An Architectural Map of the Cloud Landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD'09)*, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[9] H. Liu. Rapid Application Configuration in Amazon Cloud Using Configurable Virtual Appliances. In *ACM Symposium on Applied Computing (SAC'11), Cloud Computing Track*, pages 147–154, New York, NY, USA, 2011. ACM.

[10] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3:193–205, July 2010.

[11] R. Mietzner, T. Unger, and F. Leymann. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems (OTM 2009)*, volume 5870, pages 357–364. Springer Berlin / Heidelberg, 2009.

[12] F. P. Miller, A. F. Vandome, and J. McBrewster. *Aspect-oriented Programming*. Alphascript Publishing, 2010.

[13] S. Paal, R. Kammüller, and B. Freisleben. Customizable Deployment, Composition, and Hosting of Distributed Java Applications. In *4th International Symposium on Distributed Objects and Applications (DOA'02)*. Springer, 2002.

[14] N. Radcliffe and P. Surry. Formal Memetic Algorithms. *Evolutionary Computing*, 865:1–16, 1994.

[15] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

[16] P. Sampaio, P. Ferreira, and L. Veiga. Transparent Scalability with Clustering for Java e-Science Applications. In *11th International Conference on Distributed Applications and Interoperable Systems*, pages 270–277. Springer, 2011.

[17] R. V. van Nieuwpoort, J. Maassen, G. Wrzesińska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-Based Grid Programming Environment. *Concurrency and Computation: Practice & Experience*, 17:1079–1107, 2005.

[18] J. Varia. Cloud Architectures, 2008. http://jineshvaria.s3.amazonaws.com/public/-cloudarchitectures-varia.pdf.

[19] C. Vecchiola, X. Chu, and R. Buyya. Aneka: A Software Platform for .NET-based Cloud Computing. *Computing Research Repository*, 2009.

[20] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Wiley Software Patterns Series, 2004.

[21] J. Yan Liu. Performance and Scalability Measurement of COTS EJB Technology. In *14th Symposium on Computer Architecture and High Performance Computing*, pages 212–219, 2002.