A Framework and Middleware for Application-Level Cloud Bursting on Top of Infrastructure-as-a-Service Clouds

Philipp Leitner[1], Zabolotnyi Rostyslav[1], Alessio Gambi[1,2], Schahram Dustdar[1]

[1]Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

[2]Faculty of Informatics
Universita della Svizzera Italiana (USI)
via Giuseppe Buffi 13, 6900, Lugano, Switzerland
alessio.gambi@usi.ch

Abstract

A core idea of cloud computing is elasticity, i.e., enabling applications to adapt to varying load by dynamically acquiring and releasing cloud resources. One concrete realization is cloud bursting, which is the migration of applications or parts of applications running in a private cloud to a public cloud to cover load spikes. Actually building a cloud bursting enabled application is not trivial. In this paper, we introduce a reference model and middleware realization for Cloud bursting, thus enabling elastic applications to run across the boundaries of different Cloud infrastructures. In particular, we extend our previous work on application-level elasticity in single clouds to multiple clouds, and apply it to implement an hybrid cloud model that combines good utilization of a private cloud with the unlimited scalability of a public cloud. By means of an experimental evaluation we show the feasibility of the approach and the benefits of adopting Cloud bursting in hybrid cloud models.

I. Introduction

Recently, the vision of cloud computing [4] has led the path to a more elastic provisioning of IT resources. Users acquire and pay only for those IT resources that are actually being utilized at this very moment. Currently, two cloud deployment models are in use: private clouds are special types of virtualized data centers, which are of limited size and fully under the control of the entity that is also using it; public clouds, on the other hand, are large services, which are typically open to the public in general. For most public clouds, very little setup formalities are necessary in order to start using them. Due to their relative size, public clouds can indeed provide the illusion of boundless resources to most applications. Today, it is mostly an either-or decision whether to deploy a new application or service in a public cloud, in a private cloud, or to even host it using a traditional server. Cloud bursting, which is the idea of building an application that is able to "burst" out of a private cloud into a public cloud service as soon as the resources in the private cloud run out, is still mostly a research idea.

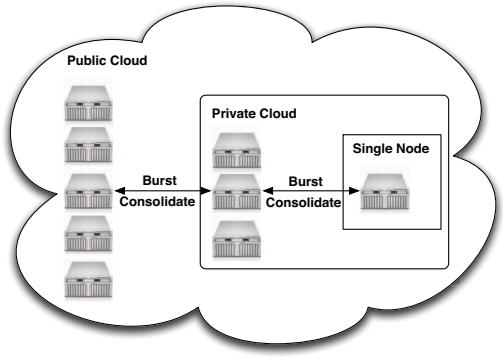

Figure 1: Basic Three-Phase Cloud Bursting Model

Figure 1 shows the reference model of cloud bursting, as considered in this paper. This model is in line with related research, for instance [8]. The reference model comprises three distinct phases. Initially, while load is very low, a single host may be sufficient to cover all requests. However, once load increases and the resources provided by the single host are not sufficient anymore, the application will burst out from the single host into the private cloud. Finally, once load increases up to the point where the private cloud runs out of resources, the

CFP13UCC-USB/13 $26.00 © 2013 IEEE
DOI 10.1109/UCC.2013.39

application bursts out into the public cloud. The public cloud will indeed, for all reasonable purposes, provide boundless resources, i.e., further bursting will typically not be necessary. Once system load goes down again, it is beneficial to consolidate again in the private cloud. We argue that deploying applications following this model allows for an efficient trade-off between cloud resource utilization and application performance. On the one hand, the model allows to use existing private resources, which are already paid for. On the other hand, the core promises of public cloud computing (boundless scalability) can still be achieved.

Unfortunately, a concrete implementation of this reference model is currently not easy. To the best of our knowledge, no standardized tools exist to support this way of application deployment. Hence, low-level implementation challenges, such as performance monitoring, code distribution, cloud management, or migration between private and public cloud, need to be solved over and over again for each development project. This severely hampers the practical adoption of cloud bursting. Our contribution in this paper is a framework for building cloud bursting applications following this reference model. Our framework transparently monitors the performance of the application in order to decide when to burst out into the public cloud, or when to consolidate again. Further, our system is able to transparently and automatically execute cloud bursting or consolidation without application outage or manual intervention. We also present a prototypical middleware that realizes the framework for Java-based applications. We numerically evaluate our middleware implementation, and show how cloud bursting impacts the performance and resource utilization of the application in comparison to other provisioning models.

II. A Framework and Middleware for Cloud Bursting Applications

Following, we introduce the main contributions of this paper.

A. Term Definitions and Background

In the most general sense, cloud bursting assumes the existence of more than one **environment** in which an application can be provisioned. The idea of co-usage of multiple environments is called **hybrid cloud**. The cloud bursting reference model, as depicted in Figure 1, comprises three environments, a single server, the private cloud, and the public cloud. All environments are interconnected either via internal network or the Internet. Further, the reference model implements the idea of **elastic computing**. An application is considered elastic *iff* it is able to expand and detract its own resource pools to accommodate current load, i.e., maximize utilization of its resources while maintaining required performance levels. Hence, elasticity can be considered a special form of the more common property of **scalability**. Under the typical assumption of boundless cloud resources [15], a single cloud already enables elasticity. However, assuming that real-life private clouds have in fact very real resource limits, there are cases where resources of multiple environments will need to be combined in order to implement a fully elastic application. Such applications **burst** into "outer" environments in times when the "inner" environment runs out of resources, and **consolidate** when the load declines again. Bursting happens as soon as the first virtual resource from an "outer" environment is used. Consolidation happens, when the last virtual resource is released back to an environment. Note that this model assumes that providers typically prefer "inner" environments over "outer" ones, and use "outer" environments mostly to cover load spikes.

To implement an elastic system, some form of **code mobility** in the agent-based computing [2] sense is necessary. Strong mobility (i.e., moving executing code including state to a different virtual resource) is preferred, but at least weak mobility (moving code without state) is required. Technically, cloud bursting is not very different to elasticity within a single environment. However, practical problems exist. For instance, significantly increased network delay is often a consideration when bursting into an outer environment. Similarly, oftentimes, data security and privacy becomes a concern as soon as a public cloud service is used to cover some requests. These issues should be taken into account when deciding on when to burst or consolidate, and which code to execute in which environment. This mapping of code executions (for simplicity referred to as **requests**) to virtual resources (and, transitively, to environments) is called **request scheduling**. Clearly, this mapping is not necessarily fixed. We refer to the process of re-assigned requests (to different virtual resources) as **replanning**. Replanning is relatively straight-forward as long as it re-assigns only requests whose processing has not yet started. However, if already executing requests should be re-assigned, application-level code **migration** becomes necessary [8]. For this, strong code mobility is required.

B. Scheduling Requests

Based on the definitions introduced in Section II-A, we now formalize the request scheduling problem. Assume that a set C of cloud environments as introduced above exists. C is ordered by preference to provider, e.g., in the reference model, $c_1 = local$, $c_2 = privateCloud$, and $c_3 = publicCloud$, with $C = \{c_1, c_2, c_3\}$. In each environ-

ment, a number of virtual resources (hosts) exists, denoted as H_c. All hosts in all environments are referred to as $H = \{H_{local}, H_{privateCloud}, H_{publicCloud}\}$. Furthermore, each environment has a maximum number of hosts that it can contain, denoted as c_{max}. For public clouds, this can be assumed to be infinitely high ($c_3 = \infty$). Hosts have a current utilization ($\sigma(h)$) as well as maximum utilization ($\sigma_{max}(h)$). In most cases, $\sigma_{max}(h)$ is identical for all hosts in the same environment, but different among hosts in different environments. We need to schedule requests $r \in R$ to hosts. Requests require resources (e.g., processing power) for their execution, denoted as $\sigma^*(r)$. Typically, we do not care in which environment a request is scheduled, but in some cases (e.g., because of data privacy), requests are only allowed to be executed in certain environments ($a(r) = C'$, with $C' \subseteq C$). Scheduling an request r now means iterating over all $c \in C$, and checking whether at least one host exists that still has sufficient free resources for r and which does not violate any location constraints of the request. Formally, the candidate set S_r of hosts consists of all hosts fulfilling the condition in Equation 1 for the request r (the function $c(h)$ denotes the environment that the host h is hosted in).

$$S_r = h \in H : \\ \sigma(h) + \sigma^*(r) \leq \sigma_{max}(h) \ \wedge \ c(h) \in a(r) \tag{1}$$

If S_r contains multiple hosts ($|S_r| > 1$), we prefer the one in the inner-most environment (i.e., the one whose hosting environment has the lowest index, $c(h) \rightarrow min!$). Further, we use the well-established *best fit bin packing* [5] approach to select between hosts in the same environment. Hence, we define the resource "waste" $w_{h,r}$ as in Equation 2. Esentially, $w_{h,r}$ signifies the amount of free resources on h after r has been scheduled to this host. According to best-fit bin packing, our selection criterion is to select the host that minimize this waste ($h \in S_r : w_{h,r} \rightarrow min!$).

$$w_{h,r} = \sigma_{max}(h) - \sigma(h) + \sigma^*(r) \tag{2}$$

If $S_r = \emptyset$, we find the inner-most environment where we are still able to acquire more hosts from and where we are allowed to schedule r in ($|H_c| < c_{max} \ \wedge \ c \in a(r)$), start a new host there, and schedule the request to the new host. In some cases it may happen that no host has sufficient capacity to execute the request, yet we also cannot instantiate a new host for it (for instance, because the request is not allowed to be scheduled on hosts in the public cloud). In such cases, we immediately trigger a replanning (see below) to migrate existing requests to a new host and make room for scheduling the new request to a host where it is allowed to run.

C. Replanning

Whenever requests finish, it is possible that we can improve resource utilization by replanning the mapping of requests to hosts. Furthermore, as indicated above, replanning can be triggered if S_r for a request r is empty and no new host can be acquired for this request.

Assuming a middleware that exhibits strong mobility properties, we are able to change the request-to-host mapping even for already started requests. To this end, we denote all currently running requests as $r \in R^*$. Every $r \in R^*$ has a remaining resource consumption, which we denote as $\sigma_\Delta(r)$. The goal of replanning is to free up as many resources in the "outer" environments as possible, so that we may be able to tear down resources. We assume we are able to migrate running requests, even between different cloud environments. We refer to this via an migration action m, which we denote as a 3-tuple of request, original host, and target host $< r, h_1, h_2 >$ ($r \in R$ and $h_1, h_2 \in H$). Clearly, h_2 needs to be part of S_r for r. Ultimately, replanning means solving the optimization problem of finding the set M of migration actions, so that (after migration) the amount of hosts used from the outmost environment is minimal. In addition, as migration is only senseful for requests that will not be finished in the close future, we only migrate requests that have remaining resource consumption larger than a defined threshold. Hosts that are unused after replanning are released back to the cloud.

For reasons of brevity, we do not go into detail here about how to solve the replanning problem. However, a practical consideration is that replanning needs to be executed frequently, hence, a very efficient algorithm is needed to solve the problem. Further, using a computationally expensive optimization algorithm is prohibitive for our framework. Hence, a design decision of our practical implementation was to use a simple variant of local optimization, GRASP (Greedy Randomized Adaptive Search Procedure [6]), to solve the replanning problem. GRASP has a linear computational complexity and can hence be executed frequently without inducing large details. We leave a more detailed discussion of replanning to a future publication.

D. CloudScale-Based Middleware

We implement this conceptual framework by extending our previous work on CloudScale, a middleware to enable dynamic scalability and elasticity of Java-based applications over IaaS Clouds [12]. The current release version of the CloudScale middleware is available as open source software from Google Code[1].

[1]https://code.google.com/p/cloudscale/

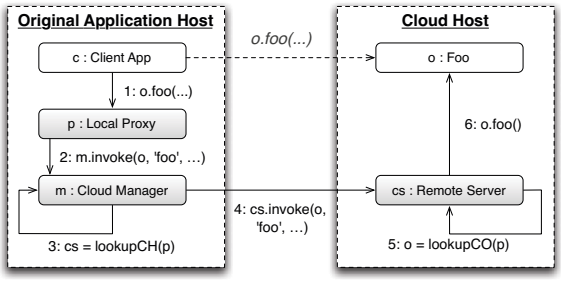

Figure 2: CloudScale Interaction Diagram (Adapted from [12])

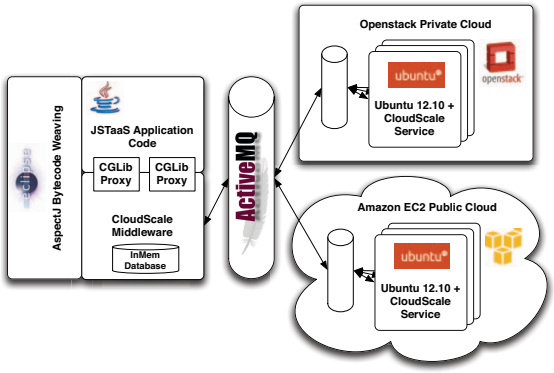

Figure 3: Implementation Overview

CloudScale uses bytecode modification to execute designated parts of an application on different hosts in the cloud. CloudScale naturally decouples Java program code and the physical location that this code is actually executed at, making it a natural fit to implement strong code mobility mechanisms. In this way, CloudScale significantly eases the development of cloud bursting solutions.

The general principle how the framework decouples code and physical location is illustrated in Figure 2. The dotted line represents the original Java source code as written by the developer (for example an usual Java method invocation such as `o.foo()`). The full lines represent what really happens after the bytecode modifications that CloudScale introduces: The original method invocation is intercepted by a proxy that finds out on which host in the cloud the object, i.e., `o`, is actually deployed, and schedules the invocation of the method `foo()` there. In this way, CloudScale is entirely transparent to the application.

The core benefit of CloudScale is that, once enabled, the middleware takes over the management of virtual resources and the software dependencies of the application [23]. Furthermore, CloudScale monitors the runtime performance of the distributed application and uses user-defined scaling policies for acquiring and releasing virtual resources, and for migrating running computations [11].

To enable Cloud bursting via CloudScale we extend the original code by implementing the logic that controls virtual machines lifecycle for the Amazon EC2[2] platform, defined additional custom scaling policy that account for deployment restrictions of public Clouds, and create customized Amazon virtual machines that automatically configure and start the CloudScale middleware as an operating system service. Figure 3 depicts the high-level architecture of system and instantiate the technological choices for the implementation. In particular, on the application host we dynamically create proxies using the Code Generation Library CGLib[3] and we use AspectJ[4] to weave in the original Java program the code that replaces user objects with our proxies. We perform this action as a post-compile step during the regular build process of the application. Together, AspectJ and CGLib allow us to shield the application developer almost entirely from the intricacies of cloud deployment and cloud bursting.

The client-side components of CloudScale communicate with the cloud resources via multiple message queues. In our current version we use Apache ActiveMQ[5] as message queue. We use the Sigar[6] framework to monitor the performance of hosts in both, the public and the private cloud. This allows us to estimate whether enough resources, e.g., processing power or free RAM, is available at a host, as required by the request scheduling model introduced in Section II.

III. Evaluation

We evaluate the feasibility of our approach and the benefits of cloud bursting over a running example. We deploy in our private cloud an application that leverages our middleware for elasticity and cloud bursting, and subject the application to a fluctuating workload that triggers elasticity across different clouds. In particular, we implement the hybrid clouds scenario where the workload fluctuations cause the elastic application to demand more resources than the ones available in the private cloud thus forcing a temporarily burst in a public one. This scenario mimics the situation of services offered by a company that *goes viral* (e.g., under a slashdot effect) and in a short time experience an increase of the service demand that might go over the capacity of the company's private cloud,

[2]http://aws.amazon.com/de/ec2/

[3]http://cglib.sourceforge.net/
[4]http://eclipse.org/aspectj/
[5]http://activemq.apache.org/
[6]https://support.hyperic.com/display/SIGAR/Home

and requires the fast acquisition of additional resources, for example from a public cloud. Once the hype is over and the service demand decreases, the private cloud of the company is sufficient again to cover the service demand. At this point, the company can release the resources from the public cloud and can run its services according to the original in-house provisioning model. In the remainder of this section we introduce the application that we use as case study to implement this scenario, and we show how our approach can enable that application to cloud burst (III-A), we describe the experimental setup adopted for the evaluation (III-B), and we conclude with a discussion of results (III-C).

A. JSTAAS: JavaScript Testing as a Service

JSTAAS is an application that provides testing of JavaScript applications as a service in the cloud. It offers a Web interface that allows clients to sign up, register a read-only Git, Subversion or Mercurial account of their application, and launch their tests periodically. JSTAAS returns the test results per e-mail after a test suite completes its execution.

We have implemented the core functionality of JSTAAS as a SOAP-based Web service using JAX-WS and Apache CXF.[7] The service receives requests for executing JavaScript test suites, executes them via the JSR-223 scripting engine, and compares the results delivered back by the scripting engine with given expected values. Then the service generates an XML-based report that summarizes the test results, and sends it to the end user by e-mail.

The parallel execution of multiple test suites is achieved by executing each of them in a separate thread, and cloud bursting is achieved by enabling CloudScale to control the JSTAAS SOAP service at runtime. We use a declarative approach implemented by means of Java annotation to let cloud scale correctly interact with the application, and Figure 4 shows the relevant code. In particular, we mark the `TestRunner` class as the atomic scaling unit of the application by means of the `CloudObject` annotation. In this way, we notify the middleware that instances of this class can be dynamically instantiated on computing nodes running in the different clouds. We also mark the `runTestSuite` method of the class with the `DestructCloudObject` to tell the middleware to run its clean-up code once the method execution is over. Furthermore, as the `TestRunner` class is also marked as serializable, the middleware can also migrate the objects between virtual machines during the executions. This example shows how with two Java annotations we enable JSTAAS to elastically distribute test executions over the multiple clouds.

```
@CloudObject
public class TestRunner implements Serializable {

  public void setupTest(Test test, TestResult result) { ... }

  @DestructCloudObject
  public void runTestSuite() { ... }

}
```

Figure 4: Integration of CloudScale and JSTAAS

B. Experimental Setup

We use Amazon EC2 as public cloud and an installation of Openstack[8] running on top of 8 Dell blade servers with two Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) and 32 GByte RAM each as private cloud. For each cloud we prepare a generic Ubuntu 12.10 cloud image running the extended implementation of the CloudScale middleware.

To showcase how the reference model of cloud bursting compares to other cloud provisioning models in terms of resource utilization and performance we run fluctuating workloads under with different setups. Each run of the experiment consists in the repeated execution of a single relatively long-running Javascript test suite. Hence, in our evaluation, we map user requests to complete execution of a test suite.

We use the following evaluation setups: (i) the `local` setup, where JSTAAS runs on a large Openstack instance (with 8 GB of RAM and 4 virtual CPUs) and does not use any additional resources. (ii) the `openstack` setup, where the core of JSTAAS is running on top of the same large Openstack instance, but test executions are scheduled using a round robin strategy over additional 20 small Openstack instances (with 1 virtual CPU each). (iii) the `ec2` setup, where we extend the `openstack` setup with 10 additional small EC2 instances (with 1 virtual CPU each). Test executions in this setup are scheduled using a round robin strategy that prefers Openstack instances over Amazon EC2 ones. (iv) the `bursting` setup, where the application is cloud bursting using our framework; at maximum, the application has access to the single large Openstack instance, 20 small Openstack instances, and up to 10 small EC2 instances, just like the `ec2` setup. The `local` setup reflects a traditional single-host provisioning model, `openstack` represents the factory-worker pattern of parallel computing that is often employed by Web services today; while `ec2` is a basic implementation of the hybrid cloud model where resources are statically provisioned.

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

[8] http://www.openstack.org/

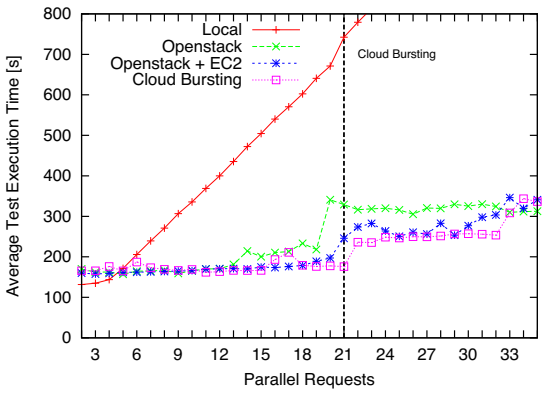

Figure 5: Test Suites Execution Time

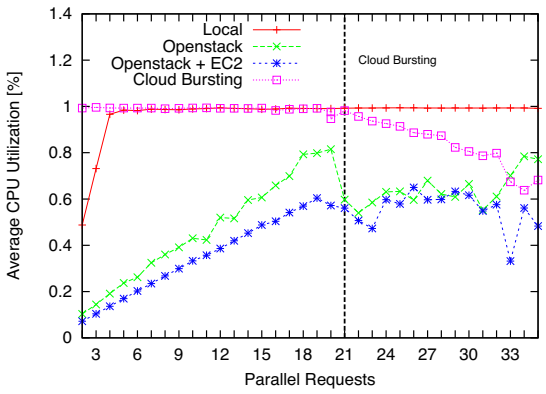

Figure 6: Average CPU Utilization of Hosts

The execution of each test suite is a CPU intensive job (the calculation of a series of large Fibonacci numbers) that is carried out by a single thread, therefore, it is not beneficial to concurrently run multiple test suite executions on the same CPU. As a consequence, given that the our maximum resource availability in terms of virtual CPUs is equal to 34, during the experiment we limit the load to vary only between 1 and 35 parallel test suite requests to avoid system saturation. For all setups we collect the service execution time, that is, the time to complete all the submitted test suites, and the CPU utilization aggregated across the running hosts. We compute the average CPU load of nodes by aggregating the CPU measurements coming from the different nodes from the beginning of the experiment to the very end. For the `local`, `openstack`, and `ec2` setups, the number of hosts that run JSTAAS is fixed, while for bursting setup the amount of running hosts changes between 1 and 31 instances according to our reference cloud bursting model.

C. Results

We report results of experiments execution across all the setups in Figure 5 and Figure 6. As reference in both figures we highlight the level of load at which JSTAAS bursts onto Amazon EC2 by means of a vertical dashed line.

In particular, Figure 5 plots the service execution time for running all the submitted test suites against the number of concurrent requests submitted to the system. This plot shows how the performance of JSTAAS evolves with respect to the intensity of the load. Figure 6 plots the average CPU usage of the system against the number of concurrent requests submitted to the system. This plot shows how the average resource utilization evolves as the load on the system changes.

Ideally, resource utilization should be as close as possible to the unity, meaning that the system resources are fully utilized during the whole experiment. At the same time, the performance of the system should be within tolerable limits, meaning that the system is doing its job well. From the plots in Figures 5 and 6 we can see that for most of the load levels the bursting setup is very close to this ideal case, and also that most of the times this setup over-perform the other ones.

Regarding the system execution time, the experiment data can be split in two phases. Between 1 and 21 parallel requests (where only private cloud resources are used), and above 21 parallel requests, where the load increases to a point where the allotted Openstack resources are not sufficient anymore. In the first phase all setups performed as expected. In the `local` setup, the system saturated fast because the number of parallel requests vastly overloaded the resources available from the single host, resulting in performance of orders of magnitude worse compared to the other setups. The performance of the `bursting`, `openstack` and `ec2` is instead comparable, as requests are scheduled on the same type of Openstack instances. In the second phase, we note a global degradation of the performance for `openstack` as computing nodes begin to saturate. The `ec2` and the `bursting` setups that exploit the additional resources leased from Amazon instead maintain more consistent performance. However, we also note that both setups suffer from slightly lower performance at this load. This is due to the EC2 instances used during the experiment not providing the same processing power as the resources from Openstack. Hence, executing the same test on those instances simply takes longer. Additionally, the more significant network latency that Amazon introduces has a negative impact on the system execution time as well. Furthermore, we note that the performance of instances in EC2 seems to be not as predictable as in Openstack (the performance curves of `bursting` and `ec2` are more "bumpy", which is in line with existing research [18]). Interestingly, this leads to the effect that at the end of

our experiment (for more than 30 parallel requests), the performance of all three setups is comparable again. All setups have overload resources for this amount of parallel requests, and gains of `bursting` and `ec2` from having access to more resources is compensated by those resources being slower.

Regarding the CPU utilization, we note the same two phases as before. In the first phase, the `bursting` setup uses its resources optimally, with very little idle time at any used processor, meaning that the system used the right amount of resources for the right amount of time. For `ec2` and `openstack`, the utilization increases the more requests have to be handled but never touches the ideal value. `ec2` has a lower average utilization, simply by virtue of having access to more resources (which are idle in this phase). In the second phase, the utilization for most setups changes in ways that are not immediately evident. Utilization of the `openstack` setup decreases as some of the resources are overloaded and need additional time to complete the request execution. During this time the other, non not-overloaded hosts, to idly wait and the value of average CPU utilization drops. For similar reasons, the utilization of the `bursting` setup decreases. In practice, the Openstack instances, which are faster than the EC ones, need to occasionally wait for the EC instances to finish the execution. Utilization in the `ec2` setup remains comparatively stable, however, it varies more, arguable due to the performance variance of Openstack and EC2 instances.

IV. Related Research

There is an existing body of research which tackles the problem of providing scalable distributed software platforms, some of which also specifically consider cloud environments. Generally, our work bears some similarity to research on cloud deployment. For instance, [14] presents a composite cloud application framework dubbed Cafe. Another example is the Rapid Application Configuration (RAC), as introduced in [13]. Both of these approaches focus on deploying an existing application to an IaaS cloud. However, unlike our work, these authors do not consider cloud bursting or elasticity at runtime. Another tool that exhibits similarities to our solution is Ibis [19]. An approach to build static hybrid clouds (comparable to our `ec2` evaluation setup) for Grid environments has been proposed in [1].

The general idea of cloud bursting was also covered in some previous work. For instance, [7] has introduced Seagull, a framework for cloud bursting applications in an enterprise setting. Unlike our solution, which aims at splitting up applications, Seagull is designed to move entire applications into and out of a public cloud. [3] introduces another framework for cloud bursting applications. Comparable to our work, they actually split up applications. However, their solution clearly focuses on data-intense applications following the map/reduce idea. Similarly, [9] has introduced a cloud bursting solution geared towards data-intense applications. The core contribution of this solution is to schedule requests either to an internal (private) cloud or to resources in the public cloud. Conversely, our framework has a more general claim. We expect that our solution is applicable to a wider range of applications. Another similarly general solution for building hybrid clouds is ANEKA [20]. While cloud bursting is not the main focus of ANEKA, this .NET-based solution is able to also support some bursting scenarios, as discussed in [21]. Finally, AppScale [10] is an implementation of Platform-as-a-Service, which also enables the construction of hybrid cloud systems. Finally, another cloud bursting solution has been proposed in [16]. The paper follows a broker-based approach, but provides little implementation details, making a truthful comparison with our work difficult.

As discussed here as well as in [8], cloud bursting often requires migration between heterogeneous clouds. Migration of virtual machines from one physical host to another can be considered state-of-the-art in today's clouds, given that many industrial-strength virtualization and private cloud solutions already support this feature. Fundamental research that led to this adoption in practice has included [17, 22]. However, [8] correctly notes that virtual machine migration is technically not useful for cloud bursting. Virtual machine migration works only within the same cloud environment, and cloud bursting by definition requires migration among several heterogeneous clouds. Hence, [8] introduces the notion of migration on application-level, a concept that we also use in our research.

V. Conclusions

We have presented a reference model, framework and middleware for building applications that are able to automatically implement cloud bursting. We have based our work on the existing CloudScale framework, which forms a suitable foundation for building automatically cloud bursting Java systems. We discussed our approach based on a realistic case study, and illustrated the effectiveness of cloud bursting via numeric experiments.

While we argue that the current incarnation of our framework is already suitable to build many cloud bursting applications, there are still a number of limiting assumptions in our approach. For instance, our current framework does not include any predictive capabilities. Bursting and consolidation decisions are currently only taken based on the current state of the application, and no estimation of

future requests is generated. This approach can lead to unwanted fluctuations if the load on the system changes often and quickly. Furthermore, CloudScale in general is currently only useful for applications where each request takes a non-trivial amount of time to finish. For applications with many, but very small, requests, the additional management overhead introduced by CloudScale can render moot all benefits achievable by cloud bursting. We are currently working on these limitations as part of our ongoing research.

Acknowledgements

This research has received funding from the Austrian Science Fund (FWF) under project reference P23313-N23, and from the Swiss National Science Foundation under contract PBTIP2-142337. Zabolotnyi Rostyslav is financially supported by the Vienna PhD School of Informatics.[9] Experimentation on Amazon EC2 was sponsored by the S.T.A.R. research group[10]) via an AWS grant.

References

[1] L. Abraham, M. A. Murphy, M. Fenn, and S. Goasguen, "Self-Provisioned Hybrid Clouds," in *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*. New York, NY, USA: ACM, 2010, pp. 161–168.

[2] L. Bettini and R. D. Nicola, "Translating Strong Mobility into Weak Mobility," in *Proceedings of the 5th International Conference on Mobile Agents (MA'01)*. London, UK, UK: Springer-Verlag, 2002, pp. 182–197.

[3] T. Bicer, D. Chiu, and G. Agrawal, "A Framework for Data-Intensive Computing with Cloud Bursting," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER'11)*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 169–177.

[4] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility," *Future Generation Computing Systems*, vol. 25, pp. 599–616, 2009.

[5] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *Approximation Algorithms for Bin Packing: a Survey*. PWS Publishing Co., 1997, pp. 46–93.

[6] T. Feo and M. Resende, "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, vol. 6, pp. 109–133, 1995.

[7] T. Guo, U. Sharma, T. Wood, S. Sahu, and P. Shenoy, "Seagull: Intelligent Cloud Bursting for Enterprise Applications," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 33–33.

[8] S. Imai, T. Chestna, and C. A. Varela, "Elastic Scalable Cloud Computing Using Application-Level Migration," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing (UCC'12)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 91–98.

[9] S. Kailasam, N. Gnanasambandam, J. Dharanipragada, and N. Sharma, "Optimizing Service Level Agreements for Autonomic Cloud Bursting Schedulers," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 285–294.

[10] C. Krintz, "The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment," *IEEE Internet Computing*, vol. 17, no. 2, pp. 72–75, Mar. 2013.

[11] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm," in *Proceedings of the 2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012.

[12] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "CloudScale: a Novel Middleware for Building Transparently Scaling Cloud Applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*. New York, NY, USA: ACM, 2012, pp. 434–440.

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

[14] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A Generic Configurable Customizable Composite Cloud Application Framework," in *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I (OTM '09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 357–364.

[15] I. A. Moschakis and H. D. Karatza, "Evaluation of Gang Scheduling Performance and Cost in a Cloud Computing System," *Journal of Supercomputing*, vol. 59, no. 2, pp. 975–992, Feb. 2012.

[16] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan, "Towards secure cloud bursting, brokerage and aggregation," in *Proceedings of the 2010 Eighth IEEE European Conference on Web Services*, ser. ECOWS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 189–196. [Online]. Available: http://dx.doi.org/10.1109/ECOWS.2010.33

[17] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'05)*. Berkeley, CA, USA: USENIX Association, 2005, pp. 25–25.

[18] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.

[19] 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*, vol. 17, pp. 1079–1107, 2005.

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

[21] C. Vecchiola, X. Chu, M. Mattess, and R. Buyya, *Aneka – Integration of Private and Public Clouds*. John Wiley & Sons, Inc., 2011, pp. 249–274.

[22] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation," in *Proceedings of the 1st International Conference on Cloud Computing (CloudCom'09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 254–265.

[23] R. Zabolotnyi, P. Leitner, and S. Dustdar, "Dynamic program code distribution in infrastructure-as-a-service clouds," in *Proceedings of the 5th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE 2013*, 2013.

[9] http://www.informatik.tuwien.ac.at/teaching/phdschool

[10] http://star.inf.usi.ch/star/