Smart Resource Allocations for Highly Adaptive Private Cloud Systems

Octavian Morariu*, Cristina Morariu, Theodor Borangiu*, Silviu Raileanu***

*University Politehnica of Bucharest, Dept. of Automation and Applied Informatics, Bucharest, Romania
E-mail: (octavian.morariu, theodor.borangiu, silviu.raileanu)@cimr.pub.ro
**CloudTroopers Intl., Cluj-Napoca, Romania
E-mail: cristina@cloudtroopers.ro

Abstract: Private cloud systems have the potential to offer the advantages of virtualization in terms of resource utilization to enterprises that can't choose migrating their data and applications outside of their premises, for legal, privacy or compliancy reasons. However, in order to achieve their full potential, these private clouds need to be extended in order to support an integrated and adaptive behaviour in regards to the specific applications that are executed on the upper layers. Modern applications are more and more aware of the fact that the underlying platform is virtualized and so resources might be allocated and de-allocated in an adaptive fashion with respect to the current load and capacity. This paper presents a service oriented mechanism for adaptability of a typical private cloud system to load fluctuations that is capable of intelligent resource allocation in both terms of amount and co-locations based on virtualization optimization. The real time monitoring information is gathered with a multi-agent system capable of multi-layer and multi-factor monitoring. The smart resource allocation is achieved with a distributed genetic algorithm that considers the workload characteristics in conjunction with physical optimum allocation and the current load. The pilot implementation is presented in the context of IBM Cloud Burst 2.1 private cloud implementation with a study on DayTrader J2EE benchmark application in load test scenarios. The results illustrate how the private cloud can show an adaptive behaviour related to the load variations.

Keywords: smart resource allocation, service oriented architecture, event driven, process, multi agent systems, genetic algorithm, cloud optimization, private cloud, adaptive provisioning.

1. INTRODUCTION

With the advances of private cloud technologies in the last half decade, an important new direction in enterprise system architectures has been established that refers to adaptability of the informational system to the dynamic requirements of the ever changing business environment. Public clouds have the advantage to benefit from a highly predictable load, when considering the large number of low granularity customers and large resource pool available, regardless if these resources are delivered as IaaS or in some sort of multi-tenant mode like PaaS or even SaaS. This predictability is assuring high resource utilization and ability for long term strategic planning for public cloud providers and this causes, to some extent, the ability to offer computing resources, services and applications over the Internet at low costs. In contrast, private clouds are characterized by a limited resource pool available, as these resources have to be physically located within the enterprise premises. These aspects cause the fluctuation in the resource demand to have a high impact on any private cloud environment and thus, it reduces its efficiency in terms of resource utilization, as it requires reservation of a significant percentage of capacity in order to be able to respect certain service level agreements (SLAs), see (Patel, 2009). This problem becomes even more important when the amount of resources utilised by the average project within the enterprise is significant in relation to the total amount of resources available in the private cloud, and when the lifetime of the project is small. These two factors cause a high fluctuation in resource allocation and thus, contribute to an overall low predictability of the entire private cloud.

In this context, in order to improve predictability of resource utilization in private cloud systems, the private cloud resource management layer needs to implement an adaptive behaviour in order to automatically adjust the resource allocation for cloud applications according to the real time capacity requirements.

Modern applications are more and more aware of the fact that the underlying platform is virtualized and so resources might be allocated and de-allocated in an adaptive fashion in regards to the current load and capacity. Active monitoring of the cloud applications at multiple layers (web, J2EE, database) as explained by (Iqbal, 2011) and recording multiple factors (CPU, memory, I/O, networking) can provide relevant information that represents complex triggers for the cloud adaptive behaviour and smart resource allocations. A similar approach for assuring platform adaptability for such applications is to extend them in order to explicitly trigger resource allocation changes depending on specific application requirements. The general idea is that the more information is provided to the private cloud system by monitoring both tools and application specific triggers, the

more accurate the adaptive behaviour will be, and so better resource utilization and better SLAs can be achieved. This addition to the basic request model resource allocation in private clouds represents a step forward in the automation of the cloud management system. This approach has direct applications in many industrial areas, where private clouds represent a valid architectural solution for replacing physical systems.

This paper presents a service oriented mechanism for assuring adaptability of a typical private cloud system to load fluctuations that is capable of intelligent resource allocation, both in terms of amount and co-locations based on virtualization optimization. The real time monitoring information is gathered with a multi-agent system capable of multi-layer and multi-factor monitoring. The smart resource allocation is achieved with a distributed genetic algorithm that considers the workload characteristics in conjunction with physical optimum allocation and the current load. The pilot implementation is presented in the context of IBM Cloud Burst 2.1 private cloud implementation with a study on DayTrader J2EE benchmark application in load test scenarios. The results illustrate how the private cloud can show an adaptive behaviour with respect to the load variations.

2. RELATED WORK

Dynamic resource allocation and adaptive behaviour of computer platforms have been studied by many researchers in both grid research and autonomous computing area. (Iqbal 2011) proposes a methodology and presents a prototype system for automatic detection and resolution of bottlenecks in multi-tier Web applications, running in cloud systems. The solution does monitor multiple tiers, but does not monitor multiple metrics, which limits the adaptability of the overall solution. Other research by (Zhu, 2011; Padala, 2007) introduce an architecture for resource allocation control within physical systems based on a two level loop. While the control mechanism is able to allocate resources with great accuracy, it lacks the capability to span over more physical machines (or blades) as available in these private cloud systems. A commercial implementation that attempts to provide a solution in this direction is (VMware DRS, 2013). This mechanism is capable to adjust the resources allocated to a single machine by using VMotion technology, based on predefined rules. While this technology is complementary to our solution, it does not address the overall application requirements that usually span over multiple workloads (virtual machines). The VMware DRS is using a virtual machine (VM) centric approach for policy definitions, and so it has a limited scope. An HP technical report by presents the possibility to dynamically allocate resources to applications in real time by use of virtualization technologies. The reports points out that allocation overhead and provisioning delay may occur due to frequent re-scheduling in the virtualization layer, which proves to be a valid point for our solution as well. The report however does not address the problem by optimal resource allocation, which, together with an effective management of events has the potential to limit the impact of this overhead. Ghanbari (2012) introduces a feedback-based optimization of resource allocation in private clouds, that focuses on mapping a given service level and resource consumption to profit metrics. Similarly (Sharma, 2011; Galante, 2012) discuss an implementation of Kingfisher method for reducing allocation costs is presented. A QoS (Quality of Service) driven model for resource allocation is discussed by (Calheiros, 2011), with emphasis on using an analytical performance queuing network model to generate intelligent provisioning instructions independent of the physical infrastructure. While these cost reduction strategies are highly effective on medium and large cloud platforms, they do have a limited applicability in small private cloud infrastructures used within a single organization. The solution proposed in this paper differentiates from the above mentioned work by relaying on real time multi-factor monitoring data and allowing definition of complex rules to model an adaptive behaviour of the private cloud.

3. WORKLOAD CLASSIFICATION AND TYPES OF TRIGGERS FOR ADAPTIVE ALLOCATION

Private clouds usually have a very specific designated purpose and can benefit of additional upfront information, decision power and flexibility at a scheduling level and, based on this, optimized schedules can be constructed. The additional information refers to the typical workload types or workload scheduling patterns while decision power and flexibility refers to the fact that in some scenarios, the customer does not require a specific time frame for the workload execution, and allows the cloud provider to schedule the workload based on a set of given rules. Applications of such clouds are in e-Learning, (Morariu, 2012), software testing, (Riungu, 2010), data mining and reporting, payroll, proof of concepts and many others as presented by (*Cloud Computing Use Case Discussion Group*, 2010). An optimum schedule of workload executions must take into consideration the primary factors that affect performance in a virtualized environment and the dependency between them. A study on this topic by (Huber, 2011; Noorshams, 2013) explores the virtualization overhead on CPU, memory and IO performance. This overhead can be lowered by grouping specific types of workloads together based on the requirements of the workload types considered (e.g. CPU intensive, IO intensive, etc.) keeping in mind the virtualization overhead between them. The result of the study shows that there is almost no influence when running IO intensive workloads together with CPU intensive workloads, while the influence between similar workloads is linear in relation with the number of such workloads.

Another important aspect to consider when scheduling workloads is to benefit as much as possible of the underlying virtualization platform features, see (Li, 2013). One such feature is memory over-commitment as supported by VMware hypervisor described in a VMware whitepaper by (VMware, 2011; Banerjee, 2013). Memory over-commitment technology allows allocation of more memory to the virtual machines than is actually available in the physical machine, if the virtual machines are running similar operating systems. The technology works by loading static sections of the guest operating system only once and allows the usage of that

memory by all the guest virtual machines. In order to benefit of this feature, it is therefore optimum to schedule similar workloads in terms of operating systems, at the same time on a physical server. The IO profile of a workload has a great impact on the virtualization performance. Current research by (Ahmad, 2012) in this area is focusing on increasing the virtual adaptors IOPS throughput and at the same time minimizing the CPU overhead generated by interrupts processing. The principle is to use a set of interrupt coalescing techniques to assure an upper level of IO notifications to the guest operating systems.

3.1 Workload Classification

The characteristics presented above have created the need for a scheduling algorithm that would consider the requirements of workload scheduling and virtualization optimization (memory over-commit, uniform distribution of IOPS). Table 1 describes the workload types considered based on: operating system (OS), CPU profile and IO profile.

Table 1. Workload profiles.

OS	CPU Profile	IO Profile
Windows	High	Low
Windows	Low	High
Linux	High	Low
Linux	Low	High

For example, a workload can run parallel processing applications which have a CPU intensive profile or run a high end relational database system, which has an IO intensive profile. Similarly the operating system used can differ according to each customer. In real life implementations, the applications are typically designed in three layers: user interface (UI) layer, business layer and database layer. When running such applications in cloud environments, the mapping with virtual machines is at the UI/Business layers, where the clustering is configured at the application server layer and at the database backend. Out of these, the application server workloads are the ones that drive the application capacity most of the time. The application server workloads show a high CPU profile, while the database workloads have rather a high IO Profile. However, these characteristics depend greatly on the application implementation and so a baseline should be established before a workload profile can be defined.

3.2 Types of triggers for adaptive allocation

The events that affect the adaptive resource allocation process can be triggered from two main sources: the monitoring layer and the application itself. The monitoring layer considered in this paper is part of our previous work described by (Morariu, 2012). This monitoring solution is using a multi agent system to gather real time data at multiple layers in the cloud application stack: hypervisor layer, OS layer, application layer (Morariu, 2013) and can be extended with custom monitoring data. The data generated by the monitoring solution sends multi-factor data (CPU, Memory, IO, and network) towards a repository in a continuum stream. This allows the possibility to define and implement central monitoring agent that could trigger events based on a set of rules, either on a single metric or on complex multi-factor conditions. This agent can use constraint programming techniques to trigger events, like IBM ILOG or similar technologies. The following table (Table 2) presents the main types of event triggers that can be defined to invoke the adaptive resource allocation process.

Table 2. Event triggers.

Metric	Definition	Type
CPU (OS Layer)	Threshold	Simple
Memory (OS Layer)	Threshold	Simple
Disk IO (OS Layer)	Threshold	Simple
Network (OS Layer)	Threshold	Simple
Combined (OS Layer)	Rule	Complex
CPU (App Layer)	Rule	Simple
Memory (App Layer)	Rule	Simple
Disk IO (App Layer)	Rule	Simple
Network (App Layer)	Rule	Simple
Application Defined	Rule	Simple
Combined (App Layer)	Rule	Complex

As shown in Table 2, the event can be triggered either by simple threshold based conditions, for example when CPU usage on a given VM is higher than a given value, or by pre-defined rules. A rule definition is essentially an XML file, with a simple schema, that supports nested definitions of rules consisting of operands, comparators and logic operators. A rule has a name, a comparator and two operands. The comparators can be HIGHER or LOWER and the operands can be a metric or a constant. An example of such a rule is presented in the outline below:

```
<rule name="high_cpu_mem">
      <operator>AND</operator>
      <rule name="high_cpu">
        <comparator>HIGHER</comparator>
        <operand>cpu_os_prc</operand>
        <operand>50</operand>
      <rule>
      <rule name="high_mem">
        <comparator>HIGHER</comparator>
        <operand>cpu_mem_prc</operand>
        <operand>70</operand>
      <rule>
</rule>
```

The association between the rules and the events triggered is stored in an XML file in the Adaptive Rules repository. An

example of such an association is presented in the outline below:

```
<association>
    <event>trigger_horizonal_add</event>
    <rule>high_cpu_mem</rule>
</association>
```

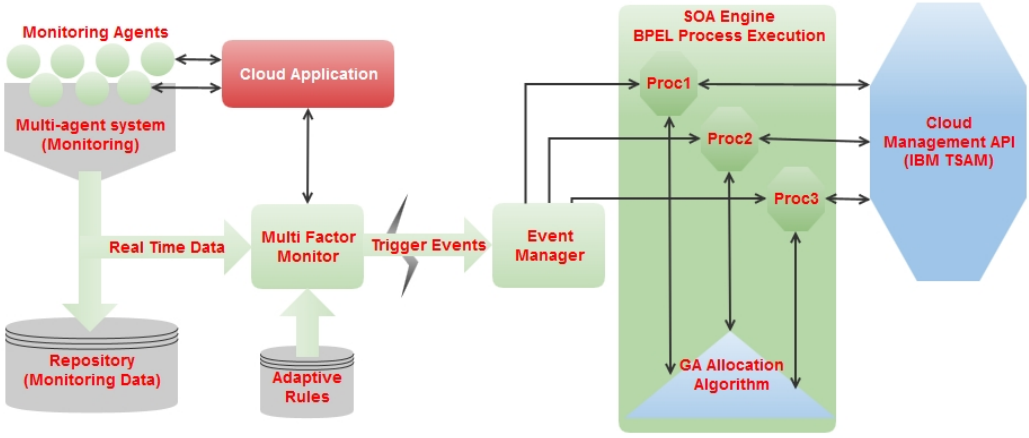

Fig. 1. Adaptive provisioning mechanism architecture.

These rules and data structures are defined by the cloud engineers and adjusted during runtime, based on the application-specific requirements.

4. ADAPTIVE PROVISIONING MECHANISM FOR OPTIMAL RESOUCE ALLOCATION

The overall architecture of the adaptive provisioning mechanism with optimal resource allocation is illustrated in Fig. 1. The mechanism consists in a comprehensive multi agent monitoring solution that is capable to gather real time metrics, a multi-factor monitor that triggers reconfiguration events based on administrator-defined rules, a business process that manages the life cycle of the cloud project and a genetic algorithm for optimal resource allocation in the private cloud. The next sections present the administrative units which are used in private clouds and considered in this implementation, they also describe in detail each component implementation and the links and bindings between them.

4.1 Administrative units

Private clouds typically use the following administrative units for resource allocation to internal customers:

- Organization - represents the organization unit that is an internal customer for the private cloud. The organization consists in a group of employees, a department, external contractors or a combination of these;
- Project - represents a logical grouping of the resources used by an organization in the context of a specific scope. An organization can have one or multiple cloud projects running in parallel;
- User - a specific user, member of a given organization. The *user* administrative unit is specifically important at higher level cloud delivery models, like SaaS, where the chargeback is based on a fixed license fee for each user;
- Virtual CPU - represents a virtual CPU which the cloud implementation can allocate to a VM. The virtual CPU (vCPU) granularity is given by the number of cores in the physical CPUs used in the cloud implementation. However, this is not always the case, as some specialised hypervisors can adjust to smaller granularities;
- Virtual Memory - represents the virtual RAM memory allocated to a VM. This memory is generally mapped to a percentage of physical RAM installed on the respective blade server, and is subject to various optimization techniques implemented by the hypervisor;
- Virtual Network - represents the virtualized network connection mapped to a VM. This virtual network connections are managed by the cloud software, usually in the context of an organization;
- Resource Allocation Request - is the standard interaction model between the customer organization and the cloud administrators in the context of resource allocations.

4.2 Adaptive provisioning mechanism architecture

The adaptive provisioning mechanism illustrated in Fig. 1 aims to scale up and down the resources required by the application, being aware of three important questions: “when to scale?”, “what to scale?” and “how to scale?”

The solution is implemented as a BPEL process and runs on top of a generic SOA engine. The overall flow is:

Step1: A new project is created in the cloud management system. The monitoring agent instantiates a new process instance, which will be running for the entire life cycle of the cloud project;
Step2: The BPEL process is initialized and starts to listen for adaptive scaling events triggered by the Event Manager based on thresholds or rules;
Step3: The multi-factor monitoring agent matches on a threshold trigger or a predefined rule and triggers an event;
Step4: The event is sent to the BPEL process that computes the scale up/down requirements based on the event received;
Step5: The genetic algorithm is invoked to determine the optimal resource allocation/de-allocation based on the current status of the cloud;
Step6: The process continues until the project is finished.

In this architecture the Event Manager has the responsibility to match the event triggered with the corresponding BPEL

process and to sequence the events that might be triggered for the same process in short amounts of time. This aspect has been proved to be very important in reducing the impact of the resource allocation overhead and resource configuration on the decision making process. It also has an important role in eliminating duplicate events that can be generated from a local monitoring perspective. These events can be generated as a result of persistent CPU usage.

4.3 Business process for project lifecycle

The business process overall view is illustrated in Fig. 2. The business process is exposed as a web service end point called **SmartAllocationServiceEndPoint** (shown in the left side of the figure). The process starts with a **receiveInput** activity that parses the payload containing the project identifier, the organization and the event actions. The values from the payload are assigned to the process variables in **Assign_Parameters** activity. At this point the process starts to listen for events in a while loop; this behaviour will continue for the entire life cycle of the project. When an event is received, the **OnMessage** activity will be triggered and the event is processed by the **HandleMessage** block (detailed in Fig. 3). The **callbackClient** informs the caller, in this case the agent invoking the web service end point, of the process completion. The **Terminate** activity finalizes the process in the BPEL execution engine.

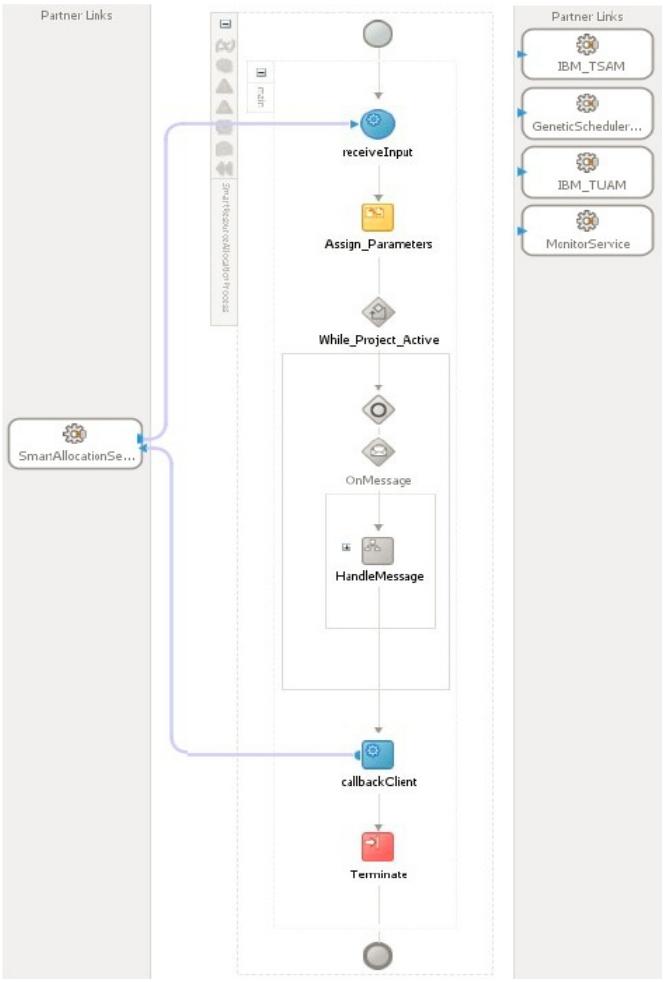

Fig. 2. Adaptive allocation BPEL process.

The ***HandleMessage*** activity block, shown in Fig. 3, starts with a ***Decode_Message*** activity, which is implemented inside a Java embedding activity, and decodes the actual event from the packaged message. Implementation wise, the messages are stored in JMS queues by the Event Manager, each queue being specific for the process. This event JMS queue is generated at runtime and known by a naming convention by both the BPEL process and the event manager.

Once the message is decoded, the next activity is to fetch the current allocation of the VMs in the cloud. This integration is realized by invoking the cloud management web services. In our pilot implementation, the process invokes the IBM Tivoli User and Accounting Management (IBM TUAM) RESTfull web service, as it can be seen in the right side of Fig. 3. The next activity invokes the web service exposed by the multi-agent monitoring solution to fetch the current load. The current allocation and the current load, and also the event generated are used to compute the changes (scale up or down decision). Once all this information is gathered and structured, the genetic algorithm is invoked to compute the best allocation pattern for the event. The design and implementation of this genetic algorithm is detailed in the next section. Once the decision on what and how to provision is in place, the InvokeCloudAPI activity invokes the cloud API to perform the resource allocation changes. In our pilot implementation the cloud API is offered by IBM Tivoli Service Automation Manager (IBM TSAM) as a set of REST web services.

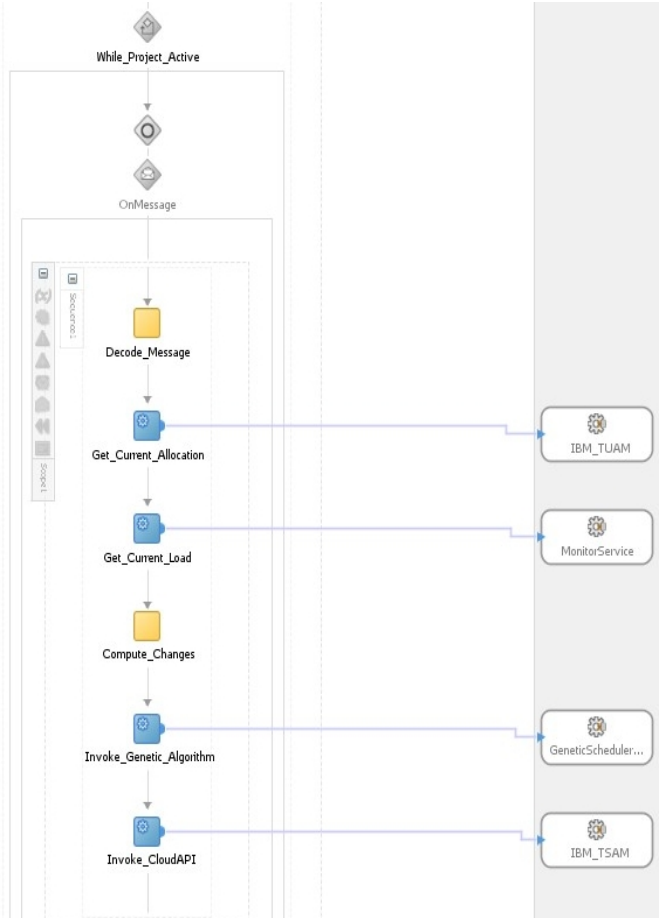

Fig. 3. HandleMessage activity block.

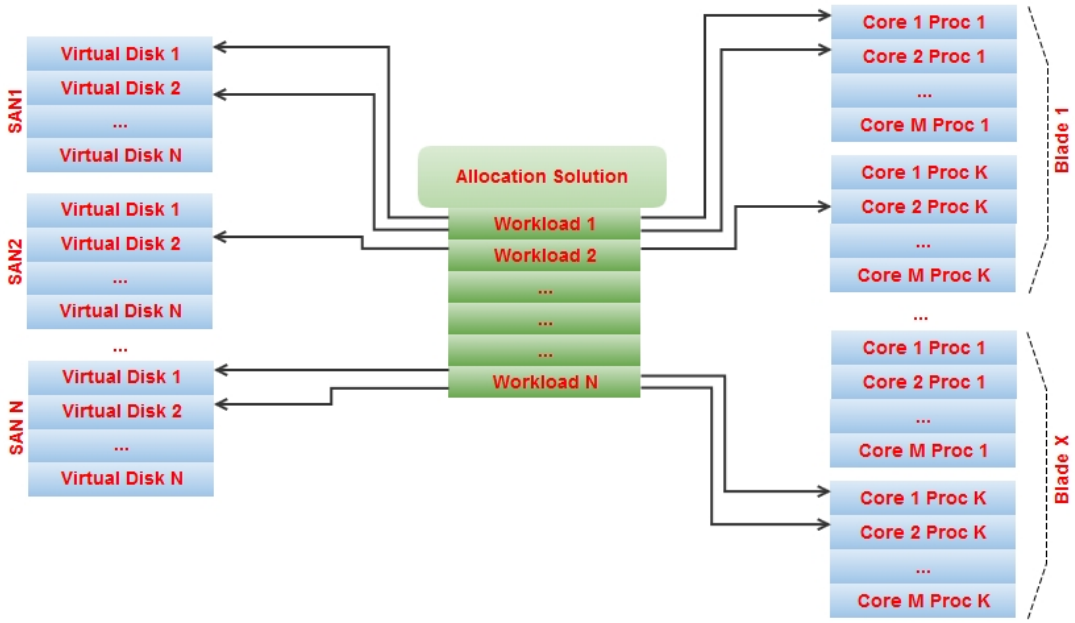

Fig. 5. AllocationSolution references (individual)

4.4 Genetic algorithm for smart resource allocation

The workload scheduling problem is NP-Complete, so the solutions generated would be sub-optimal, (Ullman, 1975). There are several hard conditions that need to be respected in order for a schedule to be accepted as a viable solution such as resource allocation levels and cloud capacity and several soft conditions that are meant to exploit the virtualization features and minimize the performance overhead of concurrent workloads.

As these conditions are sometimes conflicting, a genetic algorithm would be able to generate only sub-optimal solutions for the scheduling problem. The genetic algorithm starts with a randomly generated population of solutions and by applying operations as selection, crossover and mutation on individuals creates new generations evaluating the fitness of each individual of the population in the process. When the fitness level in the population reaches a satisfying value, a set of solutions is obtained. In the design of the genetic algorithm prototype, six objects have been considered as shown in Fig. 4:

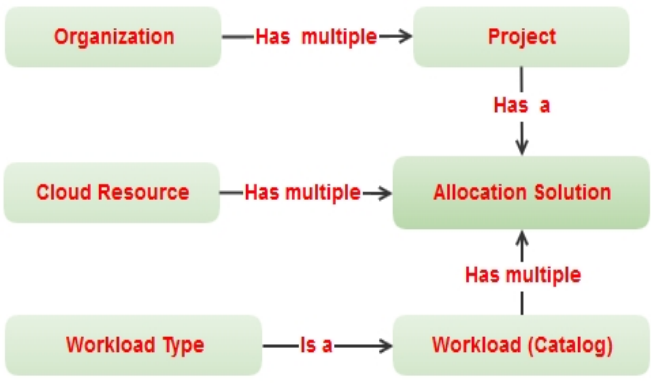

Fig. 4. Data Objects class diagram.

The AllocationSolution class holds the allocation information of the cloud resource to the workload in the scope of a project. The allocation solution class instance has a reference to the Project for which the allocation is computed. The CloudResource class represents the top of a hierarchy of cloud resources like: blade server, memory unit, virtual core, network, disk storage and so on. These resources correspond in granularity to the hypervisor administrative units, and map directly to these. The Workload class has a reference to the cloud service catalogue and represents a virtual machine template. The resources are defined by the organization administrators in the context and scope of a project. Each workload is described by a WorkloadType as mentioned in section 3 of this paper. Finally, the Organization class holds a reference to all the projects the organization has.

The ***AllocationSolution*** instances are the actual individuals in the solutions population of the genetic algorithm. The class contains at this point references to two types of resources; specifically virtual CPUs mappings on physical cores and virtual disks mappings on SANs. The AllocationSolution is illustrated in Fig. 5. The solution, implemented in the ***GlobalSolution*** class, contains a list of allocation solutions representing the entire population (Fig. 6).

Fig. 6. Global solution and allocation solution relation.

The fitness function computes the fitness value for each individual in the population. The fitness is computed by evaluating a set of conditions against the schedule instance as follows:

- ***Condition1 (Hard Condition) Total Load:*** iterates all the time slots and computes a sum of all workloads that are scheduled for each blade. If the total load scheduled

is less than the maximum capacity of the blade, then 10 points are awarded.

- ***Condition2 (Hard Condition) Possible Allocation***: Checks that the resource allocation scheme is valid. Specifically it checks that the physical cores allocated to the workload are located on the same physical blade. If the condition is fulfilled for all workloads, then 7 points are awarded.
- ***Condition3 (Soft Condition) Memory Over-commitment***: This condition checks that for each time slot only a single set of WorkloadType is scheduled. For each blade that fulfils this condition, 3 points are awarded. When all the blades are evaluated, the score is divided by the number of blades and added to the global score.
- ***Condition4 (Soft Condition) CPU Intensive***: Iterates the physical CPUs and checks that no more than two CPU intensive WorkloadTypes are scheduled at the same time. For each physical CPU that passes this check, 1 point is awarded. The score is then divided by the number of physical CPUs and added to the global score.
- ***Condition5 (Soft Condition) IO Intensive***: Similar to condition 4 above, it checks that for each SAN there are no more than two IO intensive WorkloadTypes scheduled. For each SAN that passes this check, 1 point is awarded. The score is then divided by the SANs and added to the global score.
- ***Condition6 (Soft Condition) Uniform Distribution***: This condition computes a factor characterizing the distribution of workloads. One is interested in obtaining a uniform distribution of workloads across all blades. The factor is computed by first calculating the average number of workloads scheduled across the blades, then by evaluating the difference between the calculated average and the number of workloads scheduled. A threshold of 20 workloads is considered acceptable; so if the threshold is respected across all blades, 5 points are added to the global score.

The score computed based on above evaluation is divided to the maximum score to obtain a fitness value between 0 and 1. Genetic algorithms have typically three operations: selection, crossover and mutation, refer to (Mitchell, 1998).

The *selection* operation consists in computing the fitness value for each individual in the population and sorting the population based on the results. Then, the best 65% individuals are selected for crossover operation.

The *crossover* operation represents the combination of two ***AllocationSolution*** instances that produce an offspring. The crossover operation is implemented by generating a random number X (crossover point) between 1 and N, where N is the number of workloads. The offspring will inherit the resource allocations of the first parent from Workload1 to WorkloadX and the schedule of the second parent from WorkloadX+1 to WorkloadN. Fig. 7 represents this crossover operation.

The *mutation* operation is applied to a randomly chosen subset of individuals in each generation, and consists in rescheduling of one Workload from the data structure. The new schedule is generated by randomly assigning a new alternative set of resources (CPU cores and virtual disks) for the selected Workload instance.

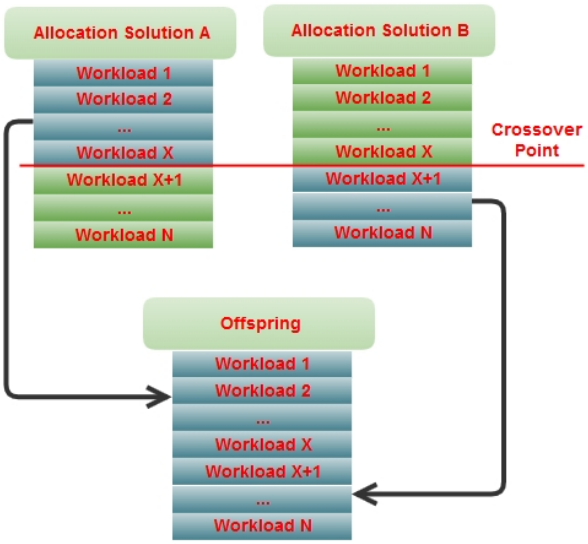

Fig. 7. Crossover operation.

Both the Workload instance and the new resource allocations are selected and generated randomly.

The genetic algorithm itself has the following structure:

```
Step1: generateInitialPopulation()
Step2: while(best individual fitness < min_fitness){
Step3:     do_crossover(best 65% individuals)
Step4:     calculate_fitness(offsprings)
Step5:     remove_worst(worst 35% individuals)
Step6:     calculate_best_individual_fitness
Step7: }
```

As one can see in the above pseudo code, the genetic algorithm sorts the individuals after the crossover operation based on the fitness, and removes the worst individuals from the population. This assures the evolution from one generation to another.

5. PILOT IMPLEMENTATION AND EXPERIMENTAL RESULTS

This section describes the pilot implementation of the adaptive resource allocation mechanism and the experimental results in a benchmark performed. The experiments are performed in three phases: the first is used to determine the monitoring baselines and monitoring system stability in regards to the upper layer application. The second phase focuses on validating the adaptive behaviour of the smart resource allocation mechanism. Finally, the third phase is focusing on testing the behaviour and characteristics of the genetic algorithm for optimal resource allocation.

5.1 Private Cloud environment and benchmark application

The cloud hardware considered for the experimental tests of the scheduling algorithm is based on an IBM CloudBurst System. The IBM Cloud Burst (IBM, 2010) system is an offering based on IBM Blade Center. It adds virtualization

using the VMware ESX hypervisor, and enhanced administration capability by leveraging the Tivoli Service Management Stack, as illustrated in Fig. 8 (IBM, 2012).The virtualization platform is powered by 14 blade servers with two Intel processors with 6 cores each at 2.8 GHz and 12 Mb L3 cache. The installed memory is 72GB for each blade. The estimated capacity is in the area of 400 concurrent virtual machines, considering an optimum scheduling with mixed CPU/IO profiles. From an integration perspective, the algorithm proposed in this paper was implemented in Java 1.6 and generates the schedule that will be provisioned to the IBM CloudBurst 2.1 solution.

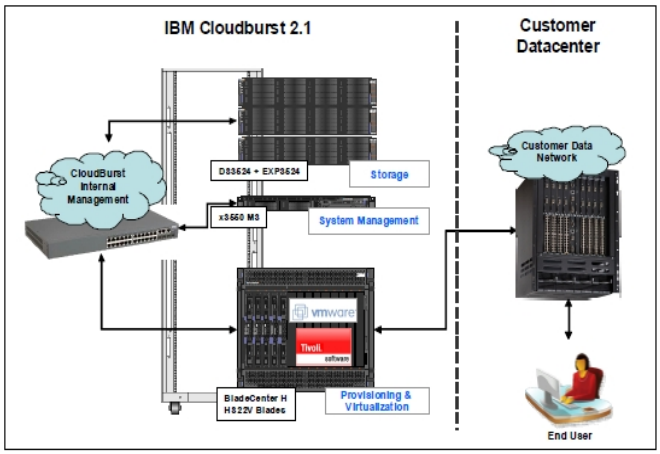

Fig. 8. IBM Cloud Burst general architecture.

The benchmark application used for testing the solution is the classic DayTrader, (Apache, 2005), application. DayTrader is a benchmark application designed to simulate an online stock trading system. The application was originally developed by IBM for WebSphere and was known as the Trade Performance Benchmark Sample. In 2005 the DayTrader application was donated by IBM to the Apache Geronimo community. The most important feature of DayTrader is that the application is easy to scale in a cluster environment by adding multiple application server nodes. The response time of the application is linear in regards to multi node scaling. This aspect is key to the test environment setup, as is important to have a clear relation between HTTP response time and the number of nodes, in order to validate the adaptive resource allocation.

The functionality implemented in the DayTrader application consists in user authentication, management of portfolio, stock quotes lookups, stock operations (buy or sell). Using load generation tools like Apache JMeter, (Halili, 2008) or HP Load Runner (Jinyuan, 2012), the workload provided by DayTrader can be used to evaluate the performance of Java Enterprise Edition (Java EE) application servers. Additionally, the application is designed to offer a set of primitives for functional and performance testing of various Java EE components in the J2EE platform and as well some common design patterns. These characteristics make DayTrader the perfect benchmark application to evaluate the capabilities of the adaptive system described in this paper.

5.2 Experimental results

The experimental environment is configured initially on a single blade server, running a single virtual machine, installed with Linux CentOS 6.4 x86-64, with two cores assigned and 16GB RAM. The application server used is WildFly 8.0.0.Beta1 (former JBoss Application Server), configured with 4GB heap size, running on Java JDK 7u45 64bit. The application was exposed to constant load from HP LoadRunner during a normal day, to validate the setup stability and establish a baseline for configuring the adaptive rules for dynamic scalability.

The first step of the experiment is represented by the validation of baseline monitoring application. For visualizing the data collected by the monitoring agents in real time, Munin was used as a frontend application to graphically show the data. Fig. 9 shows a set of collected metrics during a daylong test with constant load of the application. The metrics are collected from the guest operating system and from the application layer (the HTTP load time by day). Along with these metrics, Java heap details are collected in real time by automatic analysis of garbage collector log file. Based on these metrics, complex rules are defined for scalability. These rules create a binding between the trigger of the event, the targeted workload and the defined scaling action. The rules have a general format as defined below:

```
If <condition1> and/or <condition2> ... then
  Perform <action 1>
  Perform <action 2>
EndIf
```

For example, such a rule would be: if the CPU usage % exceeds 40% and memory usage exceeds 70%, then: allocate another virtual CPU and 1GB RAM to the virtual machine, and then configure the application server with the additional heap size. This approach represents scaling by allocating more resources to a given workload. The other approach supported is to provision another workload, which would mean scaling by clustering. The experimental setup used was configured to illustrate both approaches, defining the following rules:

```
If <HTTP Response Time > 60ms> then
  Perform <provision another cluster node>
EndIf
If <CPU_TH% > 40%> then
  Perform <provision another cluster node>
  Perform <add VCPU to VM>
EndIf
If <CPU_TH% < 30%> then
  Perform <remove a cluster node>
EndIf
If <CPU_TH% < 20%> then
  Perform <remove a cluster node and remove VCPUx2
from VM>
EndIf
```

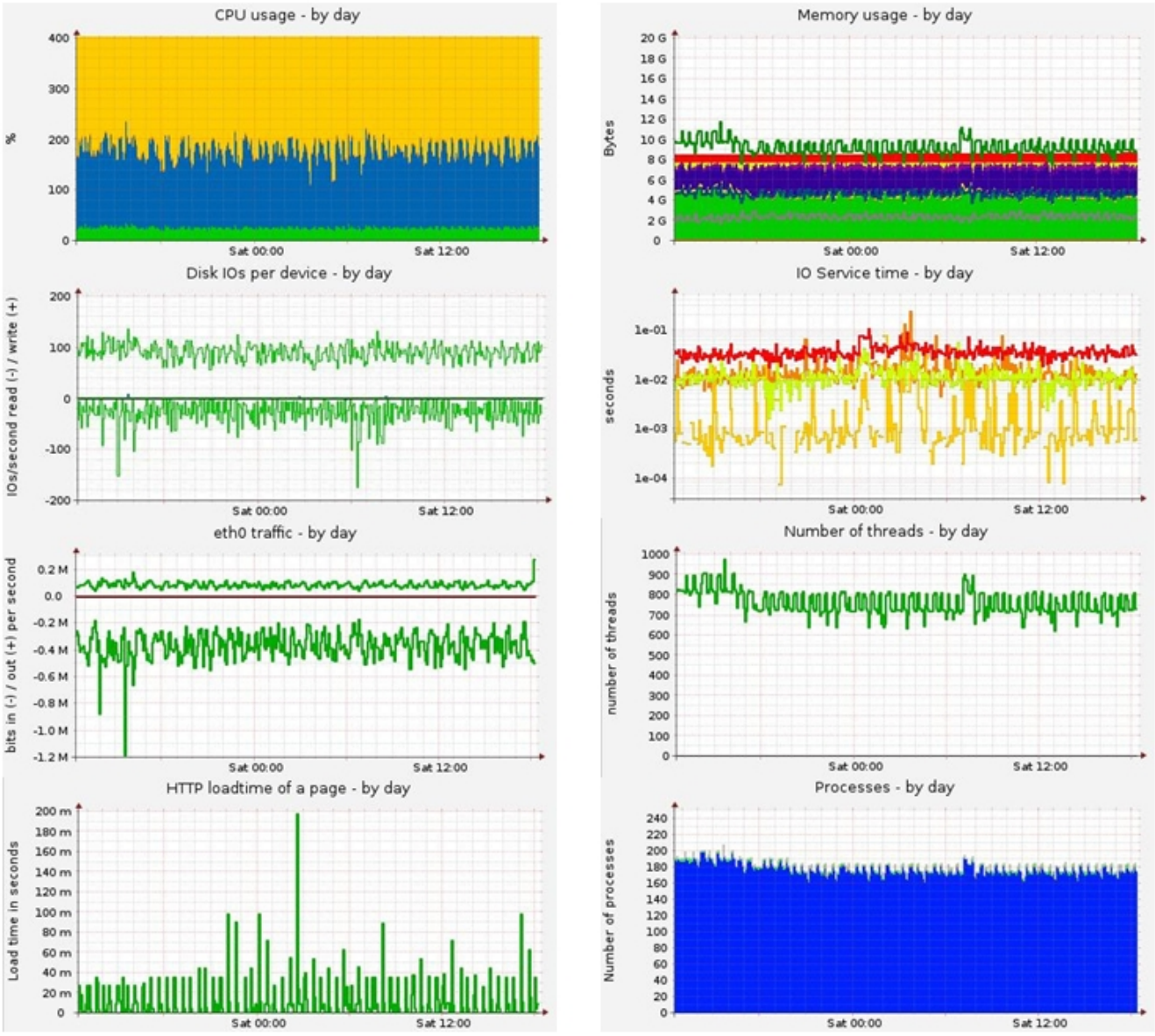

Fig. 9. Monitoring data for baseline load during a day.

The load data used is captured using Google Analytics (Clifton, 2012) on a real-life online booking application, targeted to a regional US audience, during Cyber Monday sales. This data is shown in Fig 10. As it can be seen, the load starts with less than 1000 concurrent users and peaks to 38.000 users in the early morning hours. The load fluctuates during the day and drops back to around 2000 users by the end of the day. The data was recorded from the real life application, was played with LoadRunner application against the adaptive system described in this paper and the results shown in Fig. 11 were obtained. In order to compare the traditional threshold based approach (illustrated in light green and red events) with the adaptive approach (illustrated in strong green and blue events), both were plotted on the same graph. For clarity, only the "scaling by clustering" or in other words, by provisioning additional nodes or workloads, is illustrated in Fig. 11.

The graph shows the 24 hours time-span while feeding the real data to the DayTrader application. The graph shows a slight shift with respect to the concurrent users. This is explained by the fact that initially the system utilization is low. The first events are triggered only after a relevant set of concurrent users are established. The experiment shows that the scale up is more accurate with the rule based adaptive provisioning, while the scale down is more aggressive on the threshold approach.

This experiment, obtained using the live data presented on DayTrader application, produced an improvement of 12% in resource utilization, by exploiting the adaptive approach, compared to the simple factor threshold approach supported by most solutions for automatic scaling. This has been achieved while maintaining the HTTP response time to the same levels. A second experiment was set up to validate the optimum allocation of workloads on blades using the genetic algorithm proposed in this paper.

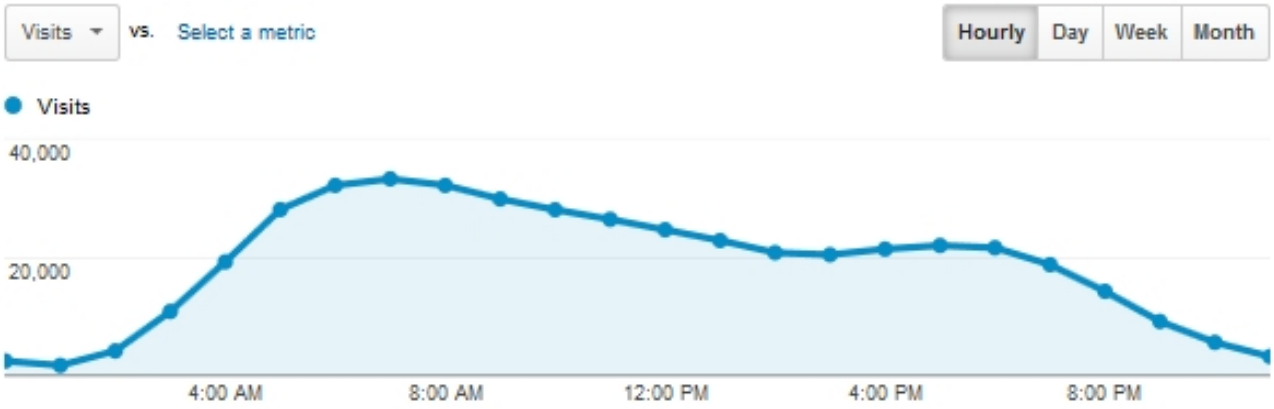

Fig. 10. Google Analytics real load during Cyber Monday.

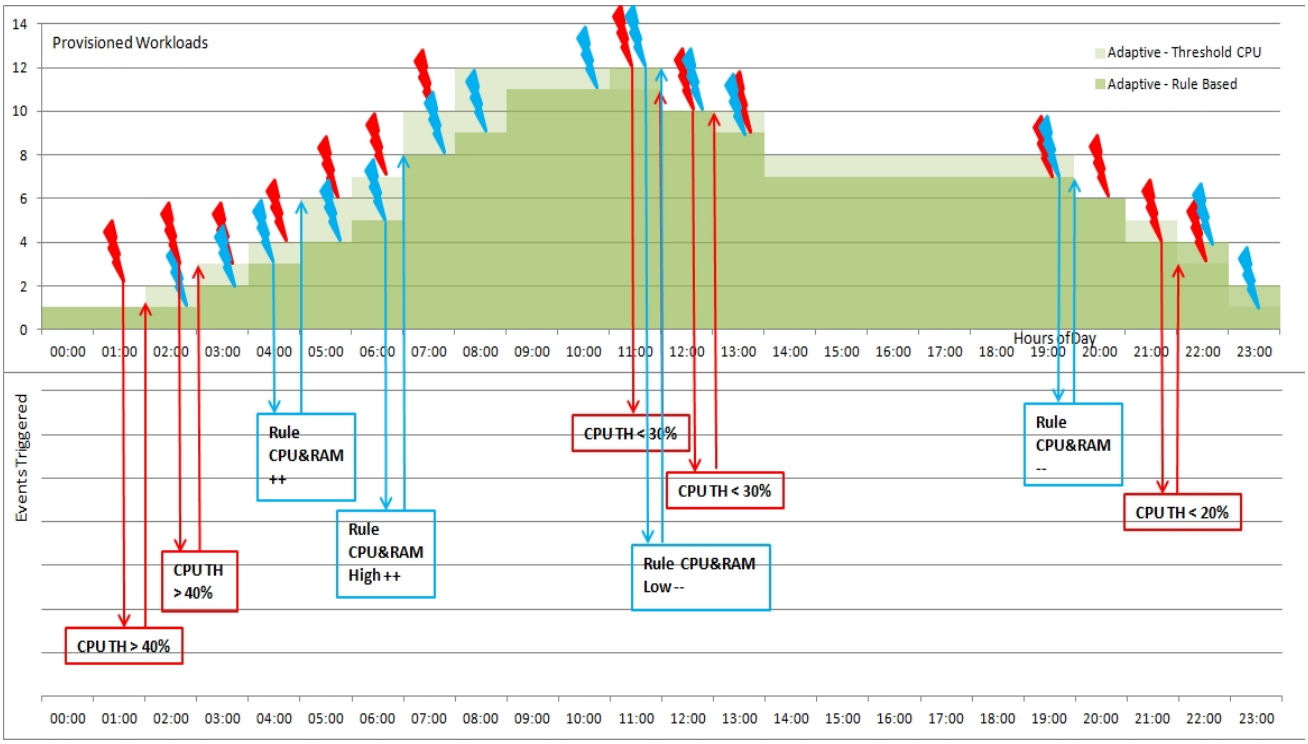

Fig. 11. Adaptive system behaviour for Cyber Monday data.

The environment consisted in random allocation of 100 workloads with mixed CPU/IO profiles using both Windows and Linux operating systems. The allocation statistics on the tracked blades were focusing on the collocations achieved with respect to the workload profiles.

Fig. 12 illustrates an almost ideal combination of workload profiles for each blade, combining CPU intensive workloads with IO intensive workloads.

The test used a uniform distribution of Windows/Linux guest operating systems. The GA allocation shows (as illustrated in Fig. 13) a tendency to collocate the similar OS workloads to favour memory over-commitment.

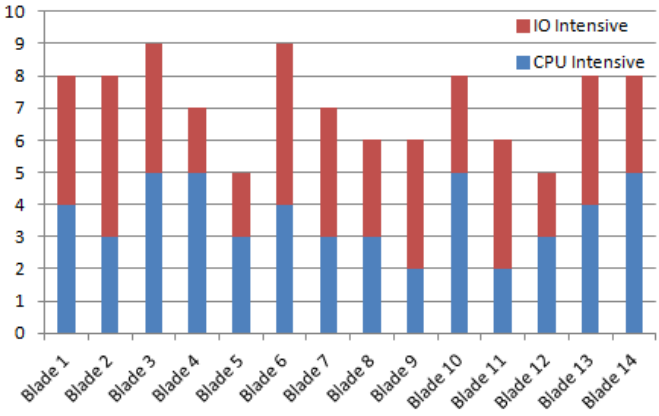

Fig. 12. CPU intensive (in blue)/IO intensive (in red).

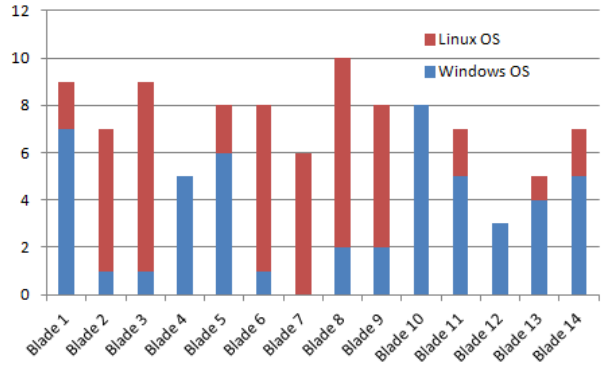

Fig. 13. Windows OS (in blue)/Linux OS (in red).

The genetic algorithm was also simulated in NetLogo (Tisue, 2004) platform in order to validate the evolutionary characteristics. Fig. 14 shows the typical fitness evolution in the population over the algorithm execution. The tests show that it takes between approximately 100 and 250 generations for the fitness function to reach the acceptable threshold, and thus, a semi-optimal solution to be selected. Fig. 15 illustrates the population diversity over the 136 generations generated in this test run.

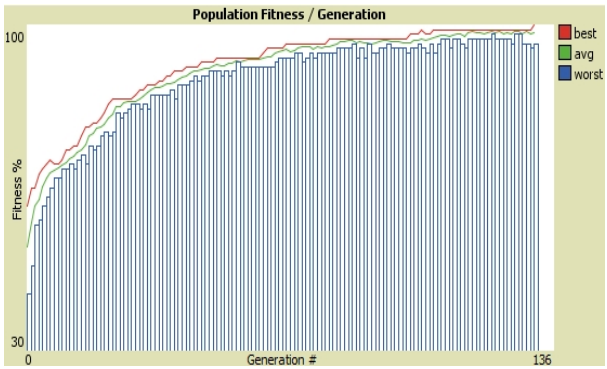

Fig. 14. Genetic algorithm fitness per generation.

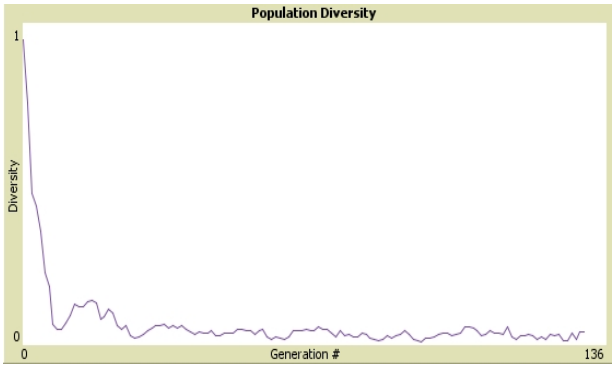

Fig. 15. Genetic algorithm diversity evolution per generation.

5.3 Limitations of the solution

One of the most important problems identified with this approach is represented by the scaling delay. The provisioning/de-provisioning of the workloads can take (in our tests) up to 5 minutes, while the start-up and configuration phase takes some additional time depending on the application.

The adaptive rules mechanism presented in this paper cannot work in real time, as the provisioning decisions will not affect the higher layer metrics immediately; the HTTP response time will be affected only after a new node is provisioned, started, configured and the load balancer configuration is changed to include this new node. In practice there are other aspects that affect this time, like cache coordination between nodes, session replication and so on. The key to resolve this problem is the Event Manager (Fig. 1), which has the role to sequence the events, and even drop duplicate events that are caused by this delay. It is essential that the BPEL process is given only unique events, at a rate that is capable to react at.

Another limitation is related to the dynamic nature of the target application running on top of the private cloud platform. The application load must not have a large variation in small amounts of time, otherwise the events will overlap. Specifically, in the pilot implementation it was established that the DayTrader application has a scaling time of around 7 minutes for provisioning an additional node, including the configuration of the clustering bindings. This has been measured with virtual machines running Linux CentOS 6.4 x86-64 with 16GB RAM. The adaptive solution is more effective for applications that scale up and down more quickly, specifically under 1 minute. These workloads are generally applications that relay on a networked file system and are used mostly for processing. In practice this means that the scaling is more effective at the application server tier, rather than at the database tier.

6. CONCLUSIONS

This paper presents a novel mechanism for adaptive resource allocation in private clouds. The mechanism is constructed as a SOA BPEL process and uses a real time monitoring solution to gather multi-factor data about the application running in the private cloud. The data collected is used to trigger re-configuration events that are handled by the BPEL process and, using a genetic algorithm, decide the optimum way to scale up and down the resources allocated.

Conceptually the mechanism supports two types of scaling: allocating more/less resources to a given workload or provisioning/de-provisioning additional workloads. This scaling decision is based on a set of flexible IF/Then/Action rules that define the trigger and link with the action to be performed. The experimental results illustrate the second approach, for clarity. The results show that the adaptive mechanism performs better (approximately 12% improvement) than a simple threshold mechanism in our test environment, based on real life usage data. However, the potential for optimizing special purpose applications where usage patterns are available is much higher, as it allows definition of complex scaling rules that can take advantage of these. Compared to other approaches for adaptive resource allocation, the solution presented in this paper is superior in two regards: the multi-factor monitoring of guest operating system and application metrics, and the optimum resource allocation based on the genetic algorithm proposed. The multi-factor monitoring allows generating complex events that provide insight information on what needs to be scaled

and when this should be done. The genetic algorithm assures the optimum scaling in this context.

Future work will extend the genetic algorithm design to consider additional resource allocation optimization criteria, including virtual network optimization and disk allocation, according to the capabilities of the private cloud.

REFERENCES

Ahmad, I., Gulati, A., Mashtizadeh, A. and M. Austruy (2012). Improving Performance with Interrupt Coalescing for Virtual Machine Disk IO in *VMware ESX Server VMware Inc*., Palo Alto, CA 94304.

Apache DayTrader Benchmark (2005). http://cwiki.apache.org/GMOxDOC20/daytrader.html, The Apache Software Foundation.

Banerjee, I., Moltmann, P., Tati, K. and R. Venkatasubramanian (2013). ESX Memory Resource Management: Transparent Page Sharing, *White paper WP-2013-01E*, available online.

Calheiros, R. N., Ranjan, R. and R. Buyya (2011). Virtual machine provisioning based on analytical performance and qos in cloud computing environments, In *IEEE International Conference on Parallel Processing* (ICPP), 295-304.

Clifton, B. (2012). *Advanced web metrics with Google Analytics*, Wiley.com.

*** Cloud Computing Use Cases (2010). A *white paper produced by the Cloud Computing Use Case Discussion Group*, Version 4.0, 2 July 2010.

Galante, G. and L. C. Bona (2012). A survey on cloud computing elasticity, In *5th IEEE International Conference* on *Utility and Cloud Computing* (UCC), 263-270.

Ghanbari, H., Simmons, B., Litoiu, M. and G. Iszlai (2012). Feedback-based optimization of a private cloud, *Future Generation Computer Systems*, 28(1), 104-111.

Halili, E. H. (2008). *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for your Websites*, Packt Publishing Ltd.

Huber, N., Von Quast, M., Hauck, M. and S. Kounev (2011). Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments, In *Proceedings of the 1st International Conference on Cloud Computing and Services Science* (CLOSER 2011), Noordwijkerhout, May 7-9, 2011, The Netherlands, SciTePress, 563 – 573.

IBM (2010). *IBM STG & Tivoli Technical White Paper*, IBM Customer Guide for CloudBurst 2.1, IBM Research Triangle Park, NC.

IBM (2012). *IBM CloudBurst on System X*, IBM RedBook.

Iqbal, W., Dailey, M. N., Carrera, D., and P. Janecek (2011). Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6), 871-879.

Jinyuan, C. (2012). The Application of Load Runner in Software Performance Test, *Computer Development & Applications*, 5, 014.

Li, J., Wang, Q., Jayasinghe, D., Park, J., Zhu, T. and C. Pu (2013). Performance Overhead Among three Hypervisors: An Experimental Study using Hadoop Benchmarks, In *IEEE International Congress on Big Data* (BigData Congress), 9-16.

Morariu, O. (2012). Resource Monitoring in Cloud Platforms with Tivoli Service Automation Management, *Proceedings of the IFAC Symposium Information Control Problems in Manufacturing* (INCOM'12), IFAC PapersOnLine, Vol. 14, part 1, 1862-1868.

Morariu, O., Morariu, C. and Th. Borangiu (2013). Transparent Real Time Monitoring for Multi-tenant J2EE Applications. *Journal of Control Engineering and Applied Informatics*, 15(4), 37-46.

Morariu, O., Morariu, C. and Th. Borangiu (2012). A genetic algorithm for workload scheduling in cloud based e-learning, In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, ACM, p. 5.

Noorshams, Q., Kounev, S. and R. Reussner (2013). Experimental evaluation of the performance-influencing factors of virtualized storage systems, In *Computer Performance Engineering*, Springer Berlin Heidelberg, 63-79.

Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A. and K. Salem (2007). Adaptive control of virtualized resources in utility computing environments, in: *EuroSys '07: Proceedings of the ACM European Conference on Computer Systems SIGOPS / EuroSys 2007*, ACM, N.Y., 289-302.

Patel, P., Ranabahu, A. and A. Sheth (2009). Service Level Agreement in cloud computing. *Cloud Workshops at OOPSLA*, 2009.

Riungu, L. M., Taipale, O. and K. Smolander (2010). Research issues for software testing in the cloud, In *2nd IEEE International Conference on Cloud Computing Technology and Science* (CloudCom), 557-564.

Sharma, U., Shenoy, P., Sahu, S. and A. Shaikh (2011). A cost-aware elasticity provisioning system for the cloud, In *31st IEEE International Conference on Distributed Computing Systems (ICDCS'11),* 559-570.

Tisue, S. and U. Wilensky (2004). NetLogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, 2004, 16-21.

Ullman, J. D. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences*, 384-393, 1975.

Wang, Z., Zhu, X., Padala, P. and S. Singhal (2007). Capacity and performance overhead in dynamic resource allocation to virtual containers, in: *Proceedings of the 10th IEEE International Symposium on Integrated Network Management*, *IM'07*, Dublin, Ireland, 149-158.

WMware, VMware Distributed Resource Scheduler DRS (2013). Online, http://www.vmware.com/products/drs/

VMware, Understanding Memory Resource Management in VMware® ESX™ Server, VMWare White Paper, 2011.

Mitchell, M. (1998). *An introduction to genetic algorithms*, A Bradford Book, 3rd printing edition, Feb. 6, 1998, ISBN-13: 978-0262631853

Zhu, X., Wang, Z. and S. Singhal (2006). Utility-driven workload management using nested control design, in: *ACC '06: American Control Conference*, Minneapolis, Minnesota, USA.