Integrating Resource Consumption and Allocation for Infrastructure Resources on-Demand

Ying Zhang, Gang Huang*, Xuanzhe Liu, and Hong Mei
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Institute of Software, School of Electronics Engineering and Computer Science, Peking University
{zhangying06, huanggang, liuxzh}@sei.pku.edu.cn meih@pku.edu.cn

Abstract—**Infrastructure resources on-demand requires resource provision (e.g., CPU and memory) to be both sufficient and necessary, which is the most important issue and a challenge in Cloud Computing. Platform as a service (PaaS) encapsulates a layer of software that includes middleware, and even development environment, and provides them as a service for building and deploying cloud applications. In PaaS, the issue of on-demand infrastructure resource management becomes more challenging due to the thousands of cloud applications that share and compete for resources simultaneously. The fundamental solution is to integrate and coordinate the resource consumption and allocation management of a cloud application. The difficulties of such a solution in PaaS are essentially how to maximize the resource utilization of an application, and how to allocate resources to guarantee adequate resource provision for the system. In this paper, we propose an approach to managing infrastructure resources in PaaS by leveraging two adaptive control loops: the resource consumption optimization loop and the resource allocation loop. The optimization loop improves the resource utilization of a cloud application via management functions provided by the corresponding middleware layers of PaaS. The allocation loop provides or reclaims appropriate amounts of resources to/from the application system while guaranteeing its performance. The two loops are integrated to run consecutively and repeatedly to provide infrastructure resources on-demand by first trying to improve resource utilization, and then allocating more resources when necessary. We implement a framework, SmartRod, to investigate our approach. The experiment on SmartRod proves its effectiveness on infrastructure resource management.**

I. INTRODUCTION

Platform as a service (PaaS) [1] such as Google App Engine [2] is emerging as a new paradigm of Cloud Computing for delivering a computing platform and solution stack as a service, which facilitates the building and deploying of cloud applications. In a cloud platform that belongs to the PaaS category, there often exist thousands of cloud applications built by different developers. These applications run simultaneously, and share and compete for infrastructure resources such as CPU and memory. Therefore, managing infrastructure resources is very important in PaaS, and the ways of acquiring and providing resources should be on-demand. Resources on-demand has a double meaning to PaaS providers [6]: (1) sufficient, the resources provided are sufficient to support the running of a cloud application; (2) necessary, the platform should not provide excessive resources to an application. Just enough resource provision can help avoid wasting resources and help save money for PaaS users (e.g., cloud application developers or owners), which is the key to the success of the pay-as-you-go nature of cloud service offerings. Additionally, although a cloud platform gives users a feeling of unlimited infrastructure resources, the resources, however, are limited. To maximize the use of the limited resources and to enable the elastic nature of Cloud Computing, a PaaS platform has to allocate resources to different applications in a time-division-multiplexed way, which after all, is to manage infrastructure resources on-demand.

Generally speaking, there are two different ways to manage infrastructure resources [7]: (1) with a given amount of resources, adjusting the configurations of an application system to improve its resource utilization (i.e., improve the system performance, because performance is the primary indicator of resource utilization); (2) providing or reclaiming appropriate amounts of resources to/from a system to satisfy its special requirements or guarantee its performance, while leaving the system configurations unchanged. The former is resource consumption management and the latter is resource allocation management [8]. In PaaS, on-demand infrastructure resource management requires resource provision to be both sufficient and necessary, and thus should integrate the above two management practices together. On the one hand, to guarantee sufficient resource provision for a cloud application, only adjusting the system configurations is not enough. An application usually cannot support various workloads under a fixed amount of resources due to its limited capability of optimizing resource utilization. Therefore, when the workload is high and the configuration-adjusting becomes inefficient, more resources should be allocated. On the other hand, resource allocation should avoid allocating excessive resources. There are two types of excessive resources: *idle* and *wasted*. The *idle* resources are brought by the static models of resource allocation. For instance, to cope with the unexpected peak workloads, a considerable amount of resources is allocated to an application once at the very start and remains unchanged even when the occurrence of peak workloads is rare. The *wasted* resources are due to the non-optimal system configurations that lead to low resource utilization. If utilization is maximized by adapting configurations to the system performance and workloads, a lot of resources will be saved. Therefore, resource allocation should be dynamic and should consider optimizing system configurations together.

There are several challenges in taking advantage of both the above two management practices in PaaS. First, the methods of resource consumption management will be many due to the large number of different cloud applications coexisting in PaaS. If adopting a specific method for managing the resource consumption of an individual application on a case by case basis, the whole management cost will be huge and even unacceptable. Thus a unified way of resource consumption

* Corresponding author

management should be found to manage applications by categories or at a specific layer in the provided software solution stacks of PaaS. Second, the dynamic resource providing and reclaiming of the resource allocation management should be done at the right time to a proper extent, and should consider the features of a cloud application itself. For instance, if memory is more critical to an application than other types of resources, it should be satisfied first when allocating resources. Third, the time to integrate the execution of the resource consumption and allocation management, and the execution frequency of each management should be adjusted dynamically according to the workload and available resources. If the adjusting is done manually, it will be time-consuming and difficult. Thus an automatic and adaptive manner is desirable to integrate these two management practices to guarantee resources on-demand in PaaS.

The nature of PaaS, however, provides possibilities to address the challenges described above. First, the middleware layers in a PaaS platform provide a convenient place for improving the resource utilization and performance of the hosted cloud applications. As we know, PaaS facilitates the building and deploying of cloud applications by delivering a computing platform and software solution stack as a service, which includes middleware (e.g., Tomcat [18], JBoss [19], and JOnAS [20]), programming APIs or SDKs (Software Development Kit) [2], and even development environment. PaaS users (e.g., cloud application developers) build their applications using these APIs or SDKs, and deploy the applications on the provided middleware layers. Considering that (1) adjusting the configurations of an individual application is complex and laborious; (2) an application uses middleware to support its functions and control the performance; and (3) it is a common practice that applications belonging to different owners occupy middleware instances or layers exclusively [14], therefore it is possible to adjust the configurations of middleware to improve the performance and resource utilization of the hosted cloud applications. Second, most middleware have already provided many management functions for adjusting configurations. For instance, Tomcat, JBoss, and JOnAS can all be managed remotely via JMX [21], and they all provide similar management functions such as increasing the size of threadpool and monitoring the number of current requests. If these functions are used properly, the system can be optimized without any source code modification or AOP-like instrumentation. Third, virtual machines (VMs, e.g., Xen [22], VMware [23], VirtualBox [24]) and virtual appliances [1] have become the main container for running cloud applications. Resource providing and reclaiming can be achieved through adjusting the resources held by a VM, thus facilitating the dynamic resource allocation management.

We can see that in PaaS, the fundamental solution to the issue of on-demand infrastructure resource management is to integrate and coordinate the resource consumption and allocation management of a cloud application. The difficulties of such a solution are essentially how to maximize the resource utilization of an application, and how to allocate resources to guarantee adequate resource provision for the system. In this paper, we use the adaptive control loop proposed in autonomic computing [9] to address this issue, because such a loop can simulate the iterative process of resource management, and can be driven automatically by policies [9]. Two adaptive control loops are introduced in our approach: the resource consumption optimization loop and the resource allocation loop. The optimization loop tries to improve the resource utilization and performance of a cloud application via management functions provided by the corresponding middleware layers of PaaS. The allocation loop provides or reclaims appropriate amounts of resources to/from the corresponding VM when the execution of the optimization loop becomes unfavorable (e.g., middleware-adjusting is inefficient for optimizing system performance). The two loops are scheduled to run consecutively and repeatedly. Given a cloud application, the optimization loop is executed repeatedly to improve resource utilization and performance until it becomes inefficient. It then requests the allocation loop to provide an appropriate amount of additional resources to the application. After that, the optimization loop is in charge of the resource management process again. If there has been a long time since the last resource-provision request arrived, the allocation loop will try to reclaim an appropriate amount of resources from the given application for conserving resources. The execution of these two loops is integrated and coordinated based on the performance of the application system, the available resources, and the SLA (Service Level Agreement) signed between the application owner and the PaaS provider. We implement a framework, SmartRod, to investigate our approach. The optimization module of SmartRod provides a collection of abstract management functions that includes two types: monitoring and adjusting. These abstract functions can be mapped to the specific management functions of a middleware. Administrators can setup performance targets of a cloud application system using the monitoring type of abstract functions. If the targets cannot be met, this module will select suitable adjusting type of abstract functions and execute their corresponding mapped functions. Thus this module can adapt to different middleware and applications, and unify optimization solutions. The allocation module of SmartRod uses the same abstraction and mapping principles to handle resource allocation to different kinds of VMs. SmartRod coordinates the execution of these two modules to guarantee infrastructure resources on-demand. The experiment on SmartRod proves its effectiveness.

The rest of this paper is organized as follows: Section II presents an example to show the importance and necessity of on-demand infrastructure resource management. Section III and IV describes our approach and implementation. Section V reports an experiment of our approach. Section VI compares our work with related efforts. Section VII concludes this paper.

II. Example

We set up a simulated PaaS environment and do an experiment to illustrate the importance and necessity of on-demand infrastructure resource management in PaaS. The cloud application used is a JEE benchmark application from the Web Performance Inc. [17], which executes the typical transactions of e-business operations (e.g., products browsing and ordering). We use Tomcat as the container of this application, and use LoadRunner [25] as the test driver. Tomcat has a configuration item called *maxThreads*, which limits the maximum number of threads used to process requests. The coming requests will be refused if their number exceeds the value of *maxThreads*. Therefore, if possible, increase *maxThreads* for improving resource utilization and system performance when there are a large number of waiting requests. LoadRunner has an adjustable parameter called *Vuser*, which determines the requests generation rate. Tomcat and the benchmark are running on a Xen VM, and LoadRunner is running on another Xen VM. The OS of both VMs is Ubuntu 9.10 server edition. The hardware configurations of the physical machines holding these VMs are CPU: Intel® Core™ Duo 2.66GHZ and Memory: 4GB.

We want to see how the performance (e.g., throughput and response time) of the benchmark is affected by the available resources and the configurations of Tomcat. Therefore, we have done the following tests:

- Test 1: allocate 256M memory to the VM hosting the benchmark; and set *maxThreads* of Tomcat to be 75.
- Test 2: allocate 256M memory to the VM hosting the benchmark; and set *maxThreads* of Tomcat to be 150.
- Test 3: allocate 384M memory to the VM hosting the benchmark; and set maxThreads of Tomcat to be 75.
- Test 4: allocate 384M memory to the VM hosting the benchmark; and set *maxThreads* of Tomcat to be 150.

As the benchmark we used is memory intensive, we allocate different amounts of memory in the tests, and fix the maximum CPU usage to be one CPU core of the dual core CPU. The workload generated is 300 *Vusers*, which increases from 0 to 300 with the rate of starting 20 *Vusers* every 10 seconds. Each *Vuser* is expected to perform a specific transaction of the benchmark application 10 times, and all the *Vusers* are scheduled to run 5 minutes. The testing results are presented in Figure 1 and 2.

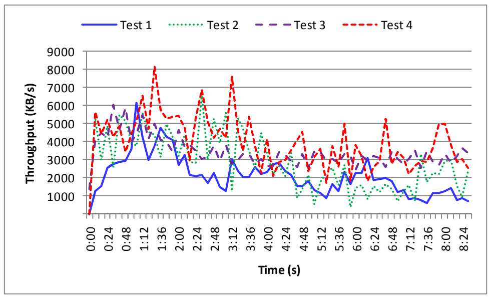


Figure 1. The throughput of each test

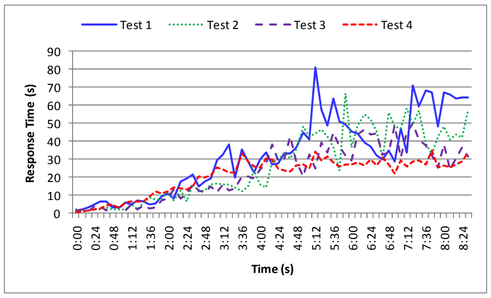


Figure 2. The response time of each test

We can see from Figure 1 that the curve of Test 4 almost always covers the curve of Test 1, which means that the throughput measured in Test 4 is larger than that in Test 1. The curves of Test 2 and Test 3 are intertwined, and they are usually sandwiched between the curves of Test 1 and Test 4, which denotes that 1) adjusting the configurations of middleware or allocating more resources to the whole system will improve the performance; 2) if configuration-adjusting and resource-allocation work together coordinately, the performance obtained will be better than that obtained by just doing one of them alone. The average throughput from Test 1 to 4 are: 1977KB/s, 2696KB/s, 3021KB/s, and 3568KB/s, which further support our view of resource management. In addition, response time of the benchmark application is also measured in our tests (see Figure 2). We can observe similar variations of curves in Figure 2 as that in Figure 1. The average response time from Test 1 to 4 are: 30.24s, 24.16s, 23.58s, and 20.32s, which also support our resource-management view.

In the above tests, configuration-adjusting and resource-allocation are done at the very start manually and statically. Although such ways of resource management can improve the system performance to some extent, they may not fully exploit the system's capability of optimizing resource utilization, and thus may usually lead to a waste of resources. For instance, we have done an additional test: Test 5, with 320M memory allocated to the VM and 300 *maxThreads* set in Tomcat. The variations of the throughput curve and the response time curve obtained in Test 5 are similar to the corresponding curves of Test 4, and each curve is intertwined with its counterpart. The average throughput of Test 5 is 3426KB/s, and the average response time is 21.24s. We can see in Test 5 that the system performance gets quite close to that in Test 4, but consuming memory 64M less than that in Test 4. The additional test tells that: 1) the resource utilization of Test 4 is not fully exploited, which leads to excessive memory allocation; 2) if possible, we can and should adjust the system configuration dynamically to improve its resource utilization and allocate resources adaptively to guarantee performance.

On-demand infrastructure resource management requires resource provision to be just enough while guaranteeing the system performance according to the SLA. When managing infrastructure resources on-demand in PaaS, a problem is: how to optimize system configurations dynamically and adaptively, and provide resources adequately? To solve this problem, we present our approach and the SmartRod framework step by step in the following sections.

III. The Integrated Control Loop-based Approach

In PaaS, infrastructure resources should be managed on-demand. Our approach uses the adaptive control loop proposed in IBM's autonomic computing white paper [9] to address this issue, because such a loop can simulate the iterative process of resource management, and can be driven automatically by policies [9]. A control loop contains four parts: (1) *monitor*, which collects data (e.g., performance indicators) from the managed system; (2) *analyze*, which processes and analyzes the collected data based on some given policies or guides to determine if a change (e.g., configuration-adjusting or additional resource-provision) is necessary; (3) *plan*, which makes a proper plan for changing; (4) *execute*, which puts the planned changes into practice. Our approach leverages two adaptive control loops: the resource consumption optimization loop and the resource allocation loop. The optimization loop tries to improve a cloud application's resource utilization and performance by executing the management functions of the corresponding middleware that hosts the application (see Section III-A). The allocation loop provides or reclaims appropriate amounts of resources to/from the corresponding VM when the execution of the optimization loop becomes unfavorable (e.g., middleware-adjusting is inefficient for optimizing system performance) (see Section III-B). The two loops are integrated to run consecutively and repeatedly to guarantee infrastructure resources on-demand by first trying to improve resource utilization, and then allocating more resources when necessary (see Figure 3 and Section III-C).

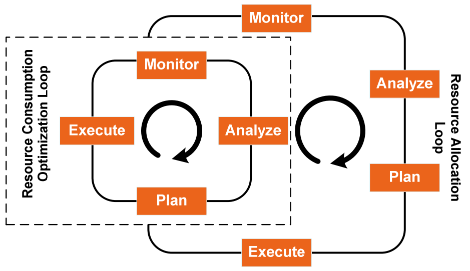


Figure 3. The Integration of the optimization and allocation loops for on-demand infrastructure resource management

A. The Resource Consumption Optimization Loop

As described in previous sections, there are usually thousands of different applications coexisting in PaaS, thus optimizing resource consumption individually per application is costly and impractical. Fortunately, as it is a common practice in PaaS that a specific type of cloud applications is deployed on a specific type of middleware, and the applications belonging to different owners often occupy middleware instances or layers exclusively, therefore resource consumption optimization in our approach is targeted at the middleware layers of PaaS. Additionally, a middleware usually provides many management functions, which can be used to monitor and adjust the performance and resource utilization of the hosted application. Thus optimizing resource consumption in the middleware layers makes our approach much more general and practical.

The goal of resource consumption optimization of a cloud application system is to improve its resource utilization while conserving resources. That is to say, improve the system performance under a given amount of resources, because performance is the primary indicator of resource utilization. Thus the optimization loop proposed in our approach is to improve the performance of a cloud application system.

The general process of performance optimization is: first, define performance-improvement goals; then select appropriate optimization scheme; and finally perform the corresponding optimization actions. As the execution of the process will consume resources (e.g., CPU or memory) more or less, thus may lead to resource competition with the normal business operations [8]. To minimize such side-effects and to simplify the process, the most commonly used way of optimization follows the "IF condition THEN action" form. In our approach, this form becomes "IF performance is unsatisfactory THEN adjusting middleware configurations". However, the performance indicators (e.g., throughput and response time) of a cloud application are unrelated to the system's business operations, which usually cannot be obtained directly from the application. Thus our approach uses the monitoring-type of management functions provided by the corresponding middleware to infer these performance values. For instance, Tomcat has provided two management items called *threadBusy* and *threadCount* respectively. If *threadBusy* is greater than 85% of *threadCount*, it usually denotes that the current workload of the hosted application is very heavy. If the situation continues, the system performance will be greatly diminished. For another instance, JBoss has provided a management item called *QueueSize*. If the value is high (e.g., greater than 100), it denotes that the application hosted on JBoss is under heavy workload. Similarly, if this situation continues, the system will exhibit poor performance. In addition to provide such monitoring-type of management functions, a middleware also provides adjusting-type of management functions for optimizing system performance. For instance, Tomcat provides a management item called *maxThreads*, which limits the maximum number of threads used to process requests. If *threadCount* approaches to *maxThreads*, and meanwhile *threadBusy* is high, then *maxThreads* should be increased appropriately to prevent Tomcat from refusing coming requests. Therefore, as long as these monitoring and adjusting types of management functions are leveraged properly, the basis of the "IF-THEN" optimization process will be easily implemented.

As described above, the optimization loop proposed in our approach is to improve system performance under a given amount of infrastructure resources. In this loop, the *monitor* part (see Figure 3) is implemented using the monitoring type of management functions provided by a middleware. For instance, the *monitor* part can include monitoring functions such as *threadCount*, *threadBusy*, *requestCount*, *errorCount*, and *maxThreads* provided by Tomcat. The *analyze* part is implemented by setting and checking the relationships between these monitoring functions or between these functions and preset thresholds. It then uses the relationships as system performance metrics. For instance, this part can check the following three relationships: *threadCount* is less than 85% of *threadBusy*; *errorCount* is less than 30% of *requestCount*; and *threadCount* is less than 95% of *maxThreads*. If one of these relationships becomes false, it usually denotes that the current system performance is unsatisfactory, so that optimization actions need to be taken. Thus we can see that the *monitor* and *analyze* parts in the loop are combined to form the "IF performance is unsatisfactory" phase of the optimization process. The *execute* part of the loop is implemented by performing the adjusting type of management functions provided by a middleware. For instance, when the system performance is considered to be unsatisfactory via monitoring and analyzing as described above. In Tomcat, the following adjusting functions can be executed: increase *maxThreads*, disable *dnsLookUp*, increase *acceptCount*, etc. These functions build up a collection of optimization actions from which appropriate functions can be selected to perform the "THEN adjusting middleware configurations" phase of the optimization process.

In the optimization loop, the *plan* part is used to select the appropriate adjusting functions and determines the execution frequencies of the selected functions. As there may be several performance metrics to be monitored and analyzed, and there are usually a collection of adjusting functions that can affect the performance, therefore, selecting appropriate adjusting functions is a challenge. The underlying problem with the selection issue is essentially a multi-objective optimization problem that is NP-hard [16]. Fortunately, there exist many algorithms and methods such as the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) or Successive Pareto Optimization (SPO) method [16] that can seek optimal solutions to such a problem. To leverage these algorithms or methods in our approach, the key is to establish a quantitative functional relationship between an adjusting function and a performance metric. Our approach uses the contribution to the improvement of a performance metric, brought by executing an adjusting function, as the quantitative relationship.

Definition: (Performance Improvement Contribution). PIC(A_i, M_j, R_t) is the contribution to the improvement of the performance metric M_j brought by executing the adjusting function A_i under a given amount of resource allocation R_t at time t.

The value of PIC is calculated as follows:

$$\mathrm{PIC}_{t+1}(A_i, M_j, R_{t+1}) = \begin{cases} \alpha \times \mathrm{PIC}_t(A_i, M_j, R_t)/2 + \beta \times |\mathrm{P}_{t+1}(M_j) - \mathrm{P}_t(M_j)| / \mathrm{P}_t(M_j) & (1) \\ \lambda \times \mathrm{PIC}_t(A_i, M_j, R_t) & (2) \end{cases}$$

In the equations, $\mathrm{P}_t(M_j)$ is the value of the performance metric M_j at time t; α, β, λ are all regulators: $\alpha + \beta = 1$, $\alpha, \beta \in (0, 1)$, $\lambda \in (0, 0.5]$. If A_i is executed at time $t+1$, then PIC_{t+1} is calculated using equation 1, else using equation 2. Let's use an example in Tomcat to illustrate how PIC is calculated. In this example, $\alpha = \beta = \lambda = 0.5$; A_i is an adjusting function that increases *maxThreads* by 10; M_j is: *threadAlert* < 0, where *threadAlert* = *threadBusy* – 0.85 × *threadCount*; and $\mathrm{P}(M_j)$ is the value of *threadAlert*. Resource allocation remains unchanged at time t and $t+1$ (R_t and R_{t+1} are both: memory 265M and CPU 2.66GHZ); At time t, PIC_t is 0.64, $\mathrm{P}_t(M_j)$ is 4.8. At time $t+1$, A_i is selected to execute, and $\mathrm{P}_{t+1}(M_j)$ is 2.4. Thus we use equation 1 to calculate the value of PIC_{t+1}, which is 0.41.

Our approach calculates the PIC values after each iteration of the optimization loop. In the next iteration of the loop, it uses these values as the input to the algorithms or methods that are leveraged in the *plan* part to help select appropriate adjusting functions (i.e., in essence, to help address a specific multi-objective optimization problem). The output of these algorithms or methods will then be transformed to generate a sequence of ordered adjusting functions such as (<A_i, 3>, <A_k, 2>, … , <A_m, 1>), which means in the *execute* part of the loop, the adjusting function A_i should be done three times, followed by executing A_k twice, and so on.

Our approach also provides a default algorithm to address the issue of selecting appropriate optimization functions. This algorithm is a simple implementation of AOF (Single Aggregate Objective) methods [16] that are often used to solve the multi-objective optimization problems. The basic idea is to combine all of the objective functions into a single functional form, called AOF. Our approach requires the system administrators to set a weight value for each performance metric. For instance, in Tomcat, assume that there exist two performance metrics: M_1 and M_2. M_1 is *threadAlert* < 0, where *threadAlert* = *threadBusy* – 0.85 × *threadCount*; and M_2 is *errorAlert* < 0, where *errorAlert* = *errorCount* – 0.3 × *requestCount*. M_1's weight is set to 3 and M_2's weight is set to 1, which denotes that if M_1 becomes false, the system performance at that time is usually worse than that when M_2 becomes false. In the default algorithm, when a performance metric M is true, its value is set to 1, else 0. The AOF of the default algorithm is: AOF = $\sum$ weight(M_i) × value(M_i). We can see that AOF_{max} is $\sum$ weight(M_i). If $\mathrm{AOF}_{current}$ is smaller than AOF_{max}, the algorithm will select the adjusting function with maximum PIC value to be executed in the *execute* part of the optimization loop.

When the optimization loop becomes unfavorable (e.g., the system performance is still unimproved after executing the loop repeatedly for three times), our approach will inform the resource allocation loop to provide more resources to the system.

B. *The Resource Allocation Loop*

As described above, the resource allocation loop proposed in our approach is to allocate appropriate amount of resources to the VM that hosts the cloud application and its supporting middleware. The main tasks of the loop are: resource providing and reclaiming. Our approach uses two algorithms to perform these two tasks, which is presented in Table 1.

We can see from Table 1 that the resource providing algorithm first waits for the resource requests issued from the nested optimization loop (see Figure 3). If the credit assigned to the system is positive, the algorithm will ask for an appropriate amount of additional resources to be allocated to the system based on the credit. After that, the algorithm will check if the resource-provision just now can actually improve the system performance. If the performance is improved, it denotes that the resource-provision is effective, thus the credit is increased to prepare for the next round of resource-provision. Otherwise, it denotes that the resource-provision is ineffective, thus the algorithm will decrease the credit to prevent from allocating excessive resources to the system.

TABLE 1. THE RESOURCE ALLOCATION ALGORITHMS

Variables and Functions:

- ***credit***: an integer variable that determines the execution of the `provideResources` function.
- ***requestsArrived***: a boolean variable that determines the execution of the `reclaimResources` function.
- **wait4Requests**(): wait until there are resource requests issued from the nested resource consumption optimization loop.
- **wait4AWhile**(): wait for some time.
- **checkPeformance()**: check if the resource-provision just now can improve the system performance.
- **provideResources**(*credit*): provide a specific amount of resources based on the credit (e.g., if *credit* is high, more resources will be provided).
- **reclaimResources**(*credit*): reclaim a specific amount of resources based on the credit (e.g., if *credit* is high, less resource will be reclaimed).

The Resource Providing Algorithm:

```
wait4Requests();
requestsArrived = true;
if (credit > 0) {
   provideResources(credit);
   boolean goodEffect = checkPerformance();
   if (goodEffect)
      credit++;
   else
      credit--;
}
```

The Resource Reclaiming Algorithm:

```
while (true) {
    wait4AWhile();
    if (!requestsArrived) {
        reclaimResources(credit);
        credit++;
    }
    requestsArrived = false;
}
```

The resource reclaiming algorithm tries to reclaim resources from the system at regular intervals. It first checks if there are resource-provision requests arrived recently. If there has been a long time since the last request arrived, the algorithm will reclaim a specific amount of resources from the system based on the credit, and increase the credit to prepare for resource-provision in the future.

Our approach requires system administrators to specify which kinds of resources need to be allocated and the reference values of allocation-amount when executing the resource allocation loop. This requirement is necessary because only the administrators know in advance that which resource may be important to the cloud application system, and the available amount of resources in the PaaS platform.

C. *The Integration of the Two Control Loops*

The optimization and allocation loops are integrated to run consecutively and repeatedly (see Figure 3). Given a cloud application, the optimization loop is executed repeatedly to improve resource utilization and performance until it becomes inefficient. It then requests the allocation loop to provide an appropriate amount of additional resources to the application. After that, the optimization loop is in charge of the resource management process again. If there has been a long time since the last resource-provision request arrived, the allocation loop will try to reclaim an appropriate amount of resources from the given application for conserving resources. The execution of these two loops is coordinated based on the performance of the application system, the available resources, and the SLA signed between the application owner and the PaaS provider. The goal of these two integrated loops is to guarantee on-demand provision of infrastructure resources in PaaS.

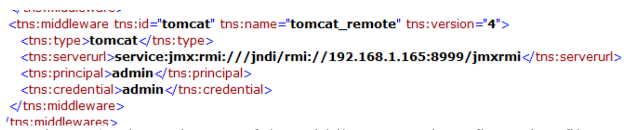
IV. THE SMARTROD FRAMEWORK

We implement a framework, SmartRod, to investigate the proposed approach. The middleware targeted in current SmartRod is java application servers such as Tomcat, JBoss, and JOnAS. The VMs supported in current SmartRod are Xen and VirtualBox. This framework contains two main modules: the RCO module and the RA module. RCO implements the resource consumption optimization loop of our approach, and RA implements the resource allocation loop.

As java application servers such as Tomcat, JBoss, and JOnAS provide management functions via JMX, thus the RCO module of SmartRod is designed to adjust these middleware's configurations for improving the system performance and resource utilization via JMX. SmartRod has provided a collection of abstract management functions that includes two types: monitoring and adjusting. These abstract functions can be mapped to the specific management functions of a middleware. For instance, in Tomcat, the management function of increasing threadpool is actually to increase the "maxThreads" attribute of the JMX MBean: "Catalina:type=ThreadPool,name=http-8080"; while in JBoss, such a function is to increase the "MaximumPoolSize" attribute of the JMX MBean: "JBoss.system:service=ThreadPool". SmartRod provides a specific class with the name "cn.edu.pku.rod.middleware.cmd.IncreaseThreadPool" to eliminate the differences. System administrators only have to specify the class name, some parameters such as the amount of threads to be increased, and the mapping attribute (e.g., Tomcat), the SmartRod framework will automatically execute the corresponding management functions of the mapped middleware (see Figure 5). Thus this module can adapt to different middleware and applications, and unify optimization solutions. Additionally, to make SmartRod extensible, it also allows users to specify their own management functions and middleware mappings to be used in SmartRod.

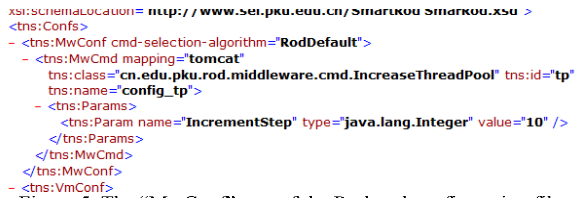
Similarly, the RA module of SmartRod uses the same abstraction and mapping principles described above to handle resource allocation to different kinds of VMs.

To leverage SmartRod, users (e.g., system administrators) only have to modify three configuration files: Middleware.xml, VM.xml, and Rod.xml. The Middleware.xml contains the mapping and connection information of the targeted middleware (see Figure 4). Similarly, the VM.xml contains the mapping and connection information of the targeted VM and VMM (Virtual Machine Monitor or hypervisor). The Rod.xml contains four parts: (1) "Monitors", which specifies the information to be collected from the targeted middleware and VM; (2) "Targets", which combines these "Monitors" to form performance metrics to be measured for determining the execution of the RCO and RA modules of SmartRod (see Figure 6); (3) "MwConf", which identifies the adjusting functions to be executed in the RCO module for improving the system performance under a given amount of resources (i.e., improving the resource utilization); (4) "VmConf", which indentifies the kinds of resources to be allocated to the corresponding VM, and the reference values of allocation-amount when the RA module is running.



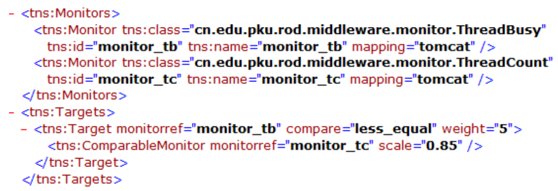
```
<tns:middleware tns:id="tomcat" tns:name="tomcat_remote" tns:version="4">
  <tns:type>tomcat</tns:type>
  <tns:serverurl>service:jmx:rmi:///jndi/rmi://192.168.1.165:8999/jmxrmi</tns:serverurl>
  <tns:principal>admin</tns:principal>
  <tns:credential>admin</tns:credential>
</tns:middleware>
/tns:middlewares>
```

Figure 4. The main part of the Middleware.xml configuration file



```
xsi:schemaLocation="http://www.sei.pku.edu.cn/SmartRod SmartRod.xsd">
<tns:Confs>
- <tns:MwConf cmd-selection-algorithm="RodDefault">
  - <tns:MwCmd mapping="tomcat"
      tns:class="cn.edu.pku.rod.middleware.cmd.IncreaseThreadPool" tns:id="tp"
      tns:name="config_tp">
    - <tns:Params>
        <tns:Param name="IncrementStep" type="java.lang.Integer" value="10" />
      </tns:Params>
    </tns:MwCmd>
  </tns:MwConf>
- <tns:VmConf>
```

Figure 5. The "MwConf" part of the Rod.xml configuration file



```
- <tns:Monitors>
    <tns:Monitor tns:class="cn.edu.pku.rod.middleware.monitor.ThreadBusy"
      tns:id="monitor_tb" tns:name="monitor_tb" mapping="tomcat" />
    <tns:Monitor tns:class="cn.edu.pku.rod.middleware.monitor.ThreadCount"
      tns:id="monitor_tc" tns:name="monitor_tc" mapping="tomcat" />
  </tns:Monitors>
- <tns:Targets>
  - <tns:Target monitorref="monitor_tb" compare="less_equal" weight="5">
      <tns:ComparableMonitor monitorref="monitor_tc" scale="0.85" />
    </tns:Target>
  </tns:Targets>
```

Figure 6. The "Monitors" and "Targets" parts of the Rod.xml configuration file

To ease the process of configuring these three files, SmartRod also provides a GUI application as an Eclipse plugin. Users can use this plugin to input the parameters for the configuration files, control SmartRod, and monitor the effects.

V. EVALUATION AND DISCUSSION

The current version of SmartRod is a research prototype under active development. We have done a simulation experiment to test its effectiveness of automatic on-demand infrastructure resource management in PaaS. The application used in the experiment is the benchmark presented in Section II. We use Tomcat as the middleware container of this application, and use LoadRunner as the test driver. Tomcat and the benchmark are running on a Xen VM, and LoadRunner is running on another Xen VM. The OS of both VMs is Ubuntu 9.10 server edition. The hardware configurations of the physical machines holding these VMs are CPU: Intel® Core™ Duo 2.66GHZ and Memory: 4GB.

We want to compare the performance and resource consumption of the benchmark application controlled by SmartRod with the counterparts of the old tests, which we have done in Section II. We chose Test 4 in the old tests as the object of comparison because the benchmark showed best performance in Test 4 compared to the other old tests (see Section II). We changed the name of Test 4 to "Without SmartRod" in the experiment to distinguish it from the new "With SmartRod" test. The following shows these two tests:

- Without SmartRod (i.e., the Test 4 in Section II): allocate 384M memory to the VM hosting the benchmark; and set *maxThreads* of Tomcat to be 150.
- With SmartRod: use SmartRod to control the resource consumption and allocation of the system. The RCO module of SmartRod runs on the VM (i.e., DomU in Xen [22]) that hosts the application and Tomcat. The RA module of SmartRod runs on the physical machine (i.e., VMM layer/Dom0 in Xen [22]) hosting the VM. The performance metric is $threadAlert < 0$, where $threadAlert = threadBusy - 0.85 \times threadCount$ (see Figure 6). The adjusting function to be executed in Tomcat is increasing threadpool by 10 each time (see Figure 4). The resources to provide and reclaim on the fly is 32M memory each time. The initial *maxThreads* is 75 and the initial memory allocated is 256M. Other resource-allocations in this test are the same as those in the "Without SmartRod" test.

In the above two tests, workloads generated by LoadRunner are the same: 300 Vusers, which increases from 0 to 300 with the rate of starting 20 Vusers every 10 seconds. Each Vuser is expected to perform a specific transaction of the benchmark application 10 times, and all the Vusers are scheduled to run 5 minutes. The testing results are presented in Figure 7, 8, and 9.

Figure 7 and 8 show that the system performance in "With SmartRod" is generally better than that in "Without SmartRod". The average throughput of the "With SmartRod" test is 3745KB/s, while "Without SmartRod" is 3568KB/s. In the "With SmartRod" test, the average response time and the 90% response time are 19.27s and 29.66s respectively, while in "Without SmartRod" test, these values are 20.32s and 30.35s respectively. These results further prove that "With SmartRod" performs better than "Without SmartRod".

Figure 9 shows that the memory consumption in "With SmartRod" (min: 256M; max: 384M; avg.: 340.18M; and ends in 352M) is generally less than that in "Without SmartRod", which is 384M. The former's threadpool size (min: 75; max: 335; avg.: 275; and ends in 335) is larger than the latter's (i.e., 150). The results tell that SmartRod indeed tries to exploit the system's capability of optimizing resource utilization for managing infrastructure resources on-demand. Additionally, what should be noted is that, in Figure 9, the threadpool size and the memory allocated are not always on the increase. It shows that (1) a system has performance bottlenecks. When the workload is above certain threshold and the resources allocated exceed certain amounts, the system performance can hardly be improved no matter how many additional resources are newly allocated or how the system is optimized; (2) SmartRod can automatically stop allocating additional resources to a system and/or stop optimizing configurations when the system experiences performance bottlenecks; (3) the cost of SmartRod is acceptable. It can be controlled due to that the thresholds of performance metrics (see Figure 6) and the resource allocation intervals/amounts (see Table 1) set in SmartRod are adjustable, and SmartRod will automatically stop working when (2) occurs.

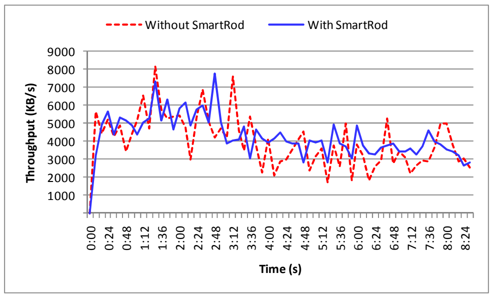


Figure 7. The throughput of the with/without SmartRod tests

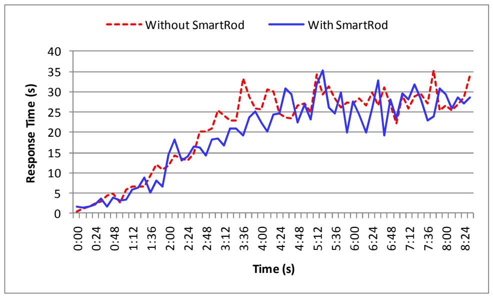


Figure 8. The response time of the with/without SmartRod tests

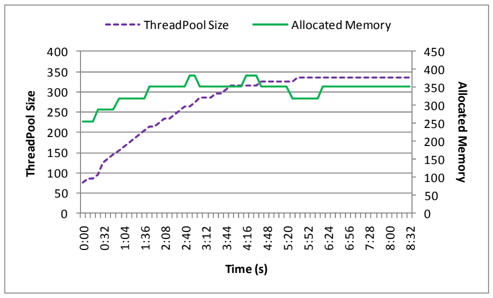


Figure 9. The changes of threadpool size and the allocated memory in the "With SmartRod" test

There are also some defects existing in the current version of SmartRod. For instance, the threadpool size can be increased by SmartRod when the workload increases, but it cannot be decreased correspondingly when the workload decreases. There is no need to maintain a large threadpool when workload is relatively low. The next release of SmartRod will address this problem by specifying or automatically inferring adjusting functions such as decreasing threadpool size as compensations, so that to enhance on-demand infrastructure resource management to be much more timely and adaptively.

VI. Related Work

Managing infrastructure resources on-demand is critical to PaaS. Zhang et al. [3] [4] has proposed a Cloud Computing Open Architecture called CCOA. There are at least two modules in CCOA that are closely related to the on-demand infrastructure resource management. They are: "Cloud IT Infrastructure Management" and "Cloud Quality and Governance". These two modules work together to provide adequate infrastructure resources to the cloud users and govern the operations of the cloud service offerings according to a set of quality indicators and normative guidance. In addition, on-demand infrastructure resource management has been thought as an enabler for the elastic nature of Cloud Computing and the way to the green revolution [10].

As described in previous sections, one basic way of on-demand infrastructure resource management in a PaaS platform is called resource consumption optimization. Several work focus on this way for improving system performance under a given amount of resources. For instance, Yang et al. [11] propose a profile-based approach to developing just-in-time scalability for cloud applications. In this approach, expert's knowledge of scaling application servers is captured as profiles. Guided by profiles, profile driver automates the scaling of applications (e.g., components constituting an application increases or decreases) to ensure performance and improve resource utilization. Jordan et al. [12] extends a JEE server (the JEE Reference Implementation) with a module called RM for managing resources flexibly in fine-grained inside the server, and provides a set of API to operate this module. Its goal is to control the resource usage and allocation among the applications hosted in the server. These approaches usually have to modify a system for improving its performance and resource utilization, while our approach uses the management functions provided by the corresponding middleware layers of PaaS for optimizing, thus leaves the application system intact.

The other basic way of on-demand infrastructure resource management is called resource allocation management. The goal is to provide adequate resources to a cloud system to satisfy its special requirements or guarantee its performance. In [8], OpenNEbula is used as an engine for on-demand resource provision. It lies between Guest OS (i.e. VM) and hypervisor (i.e. VMM) for transforming a distributed physical infrastructure into a flexible and elastic virtual infrastructure, which adapts to the changing demands of the VM workloads. In [13], the Sulcata resource management system is implemented for online virtual cluster provision. Sulcata uses a resource mapping strategy called "Load Aware Mapping" to reduce deploying overhead and balance resource utilization. In [14], a utility based approach combines the "local decision module" and the "global decision module" to determine VM resource-adjusting or migration. In [15], a two-level agent architecture is proposed. When a VM node is overload, the local agent will try to allocate more resources to it from the resources possessed by the corresponding physical node. When the physical node is also overload, one of its VM will be migrated to another underloaded physical node by the global agent, thus to guarantee sufficient resource provision. The above approaches determine resource provision mainly based on monitoring VM workloads (i.e., CPU or memory utilization). Our approach predicts the system performance by leveraging the management functions provided by the middleware layers. If the performance is considered to be unsatisfactory, our approach will first try to optimize the system configurations before allocating additional resources.

VII. Conclusion and Future Work

PaaS provides a cloud platform to enable service delivering in terms of scalable infrastructures, middleware, and even development environment [3], which facilitates the building and deploying of cloud applications. On-demand resource management is critical to PaaS due to the fact that there are thousands of cloud applications sharing and competing for infrastructure resources. Additionally, managing infrastructure resources on-demand is the key to the success of the pay-as-you-go nature of cloud service offerings, and the enabler for the elastic nature of Cloud Computing. In this paper, we have proposed an integrated control loop-based approach to managing infrastructure resources in PaaS. Two adaptive control loops are leveraged in our approach: (1) the resource consumption optimization loop, which improves the resource utilization and performance of a cloud application via management functions provided by the middleware layers of PaaS; and (2) the resource allocation loop, which provides or reclaims appropriate amounts of resources to/from the corresponding VM while guaranteeing the system performance. The two loops are integrated to run consecutively and repeatedly to provide infrastructure resources on-demand by first trying to improve resource utilization, and then allocating more resources when necessary. We implemented the SmartRod framework to investigate our approach, and the simulation experiment proved its effectiveness. Future work on SmartRod includes improving it to adapt to more middleware and VM types. We also plan to extend the optimization loop from middleware layers to OSes, and extend the allocation loop to include VM migration if necessary.

Acknowledgements

This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003, 60873060, 60933003; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z16; and the National S&T Major Project under Grant No. 2009ZX01043-002-002.

References

[1] Cloud Computing, http://en.wikipedia.org/wiki/Cloud_computing
[2] Google App Engine, http://code.google.com/appengine/
[3] Liang-Jie Zhang and Qun Zhou, CCOA: Cloud Computing Open Architecture, the IEEE International Conference on Web Services (ICWS), 2009, pp.607-616.
[4] Liang-Jie Zhang, Carl K Chang, Ephraim Feig, Robert Grossman, Business Cloud: Bringing The Power of SOA and Cloud Computing, IEEE International Conference on Services Computing (SCC 2008)
[5] Sun, Introduction to Cloud Computing Architecture, White Paper, 1st Edition, Sun Microsystem, 2009
[6] A Chandra, et al., Quantifying the Benefits of Resource Multiplexing in On-Demand Data Centers, ACM Workshop on Algorithms and Architectures for Self-Managing Systems, 2003
[7] Hector A, Gordon S, Geoff C. Adaptive resource management in middleware: a survey. IEEE Distributed Systems Online, 2004
[8] Iganacio M. Liorente, Cloud Computing for on-Demand Resource Provisioning, 7th NRENs and Grids Workshop, 2008
[9] IBM, An architectural blueprint for autonomic computing, 2006
[10] M. Armbrust et al., Above the Clouds: a Berkeley view of Cloud Computing, http://abovetheclouds.cs.berkeley.edu
[11] Yang Jie, Qie Jie, Li Ying, A Profile-Based Approach to Just-in-Time Scalability for Cloud Applications, IEEE International Conference on Cloud Computing, 2009, pp. 9-16
[12] Jordan M, et al., Extending a J2EE server with dynamic and flexible resource management, Middleware, 2004, pp. 439-458
[13] Yang Chen, Tianyu Wo, Jianxin Li, An Efficient Resource Management System for On-line Virtual Cluster Provision, IEEE International Conference on Cloud Computing, 2009, pp. 72-79
[14] Hien Nguyen Van, Frederic Dang Tran, Autonomic virtual resource management for service hosting platforms, Cloud'09, pp. 1-8
[15] Weng, et al., "Automatic Performance Tuning for the Virtualized Cluster System", in Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS), 2009, pp. 183-190
[16] MMO, http://en.wikipedia.org/wiki/Multiobjective_optimization
[17] www.webperformanceinc.com/library/reports/ServletReport/ServletBenchmark.war
[18] Tomcat, http://tomcat.apache.org/
[19] JBoss, http://www.jboss.org/
[20] JOnAS, http://jonas.ow2.org/
[21] JMX, http://java.sun.com/javase/technologies/core/javamanagement
[22] Xen, http://www.xen.org/
[23] VMware, http://www.wmware.com/
[24] VirtualBox, http://www.virtualbox.org/
[25] LoadRunner, https://h10078.www1.hp.com/