Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

OCSO: Off-the-cloud service optimization for green efficient service resource utilization

Daren Fang[1*], Xiaodong Liu[1], Lin Liu[2] and Hongji Yang[3]

Abstract

Many efforts have been made in optimizing cloud service resource management for efficient service provision and delivery, yet little research addresses how to consume the provisioned service resources efficiently. Meanwhile, typical existing resource scaling management approaches often rest on single monitor category statistics and are driven by certain threshold algorithms, they usually fail to function effectively in case of dealing with complicated and unpredictable workload patterns. Fundamentally, this is due to the inflexibility of using static monitor, threshold and scaling parameters. This paper presents Off-the-Cloud Service Optimization (OCSO), a novel user-side optimization solution which specifically deals with service resource consumption efficiency from the service consumer perspective. OCSO rests on an intelligent resource scaling algorithm which relies on multiple service monitor metrics plus dynamic threshold and scaling parameters. It can achieve proactive and continuous service optimizations for both real-world IaaS and PaaS services, through OCSO cloud service API. From the two series of experiments conducted over Amazon EC2 and ElasticBeanstalk using OCSO prototype, it is demonstrated that the proposed approach can make significant improvement over Amazon native automated service provision and scaling options, regardless of scaling up/down or in/out.

Keywords: Cloud computing; Auto scaling; Service API; Service optimization; Service resource consumption management

Introduction

Historically, the efforts made in optimizing ICT (Information Communication Technology) energy consumption have been largely focusing on efficient utilization of physical computational resources e.g., green networking, storage and computation in large scale data centers [1]. In the era of cloud computing (CC), however, green optimization should involve two sets of major objectives: green service (resource) provision [2] as well as green service (resource) consumption [3]. While the former is largely focused with a diversity of approaches proposed, the latter is seldom adequately addressed.

Statistics shows that large and complex server farms and data centers all over the world constitute the majority of global ICT energy consumption [4,5]. This attracts several attentions and results into numerous research practices. Addressing the service pool and data center resources utilization, the optimization approaches are seen as resource virtualization [6], server consolidations [7], workload consolidations [8], dynamic voltage and frequency scaling (DVFS) [9], as well as a series of optimized resource allocation and scheduling techniques. These approaches are typically designed for infrastructure owners, e.g., cloud service providers, so that they can run their own infrastructure efficiently [10]. Yet, these optimizations should seldom be regarded as achieving the ultimate energy efficiency, since they only deal with one side of the problem: the service/resource provision efficiency [11]. Currently, very few approaches try to enable service consumption optimization from the service consumer perspective. In fact, while considering the full life-cycle of cloud services/resources, the efficiency in relation to how end users utilize the provisioned services/resources also matters significantly.

For instance, Infrastructure-as-a-service (IaaS) services which provide cloud virtual machines (VMs) allow users to select customizable VM sizes (types), but if users always have to choose over-provisioned VMs and inefficiently use them (even if they only occasionally need that much of computing power), considerable reserved unused resources will be wasted. Similarly, although Platform-as-a-Service

* Correspondence: d.fang@napier.ac.uk
[1]School of Computing, Edinburgh Napier University, 10 Colinton Road, Edinburgh EH10 5DT, UK
Full list of author information is available at the end of the article

(PaaS) services provide automatic scaling options for developers to build scalable applications, it is critical that whether the automatic scaling feature would function effectively and flexibly enough to serve ultimate green efficiency requirements while experiencing extreme workload dynamics.

The key factors that result into the above inefficient service resource utilizations are considered twofold: I) Green efficiency is only one of the many criteria that service providers concern; it is hardly taken as the primary key factor [12]. II) Due to a wide range of reasons such as security, privacy, audition, users are typically left with limited control and customizability over a great deal of service configuration parameters [13]. Nonetheless, nowadays, many service providers offer advanced granular service manipulation through service APIs (Application Programming Interfaces), which actually enables a possible means of third-party optimization solutions: Firstly, customized API requests would provide advanced access and control to cloud services (resources) from a much lower level. This can be used for implementing appropriate service customizations towards the green efficiency requirements. Secondly, with proper efficiency-aware algorithms which are controlled by some user-specified optimization parameters, a user-side service optimization would be a promising alternative to address service efficiency issues via the greener service consumption.

To fill the above research gaps, this paper proposes Off-the-Cloud Service Optimization (OCSO) - a novel user-side service consumption optimization approach towards ultimate energy-efferent resource utilizations. OCSO enables IaaS and PaaS users to manipulate service resources utilizations so that the optimized service instances can scale up/down or in/out automatically and intelligently using OCSO cloud service API. The contributions of the work are: 1) A heuristic-based off-the-cloud green cloud service optimization approach which enables customizable and intelligent scaling control by using dynamic green boundaries and thresholds according to multiple monitor categories data for the excessive high/low workload volumes encountered; 2) An IaaS-specific resource consumption optimization approach which can scale VMs up/down proactively by transiting the inefficient running VMs to their successors at appropriate VM sizes and reallocating their workloads to them; 3) A PaaS-specific resource consumption optimization approach which can scale VMs in/out effectively by calculating the optimal number of VMs needed to provision for dynamically varied workloads and then facilitating application platform environment modifications on-the-fly.

The rest of the paper is organized as follows: Section "Related work" discusses the related research of service optimization regarding optimized resource scheduling and scaling approaches, as well as a series of well-known CC industry solutions. Section "OCSO system architecture" describes the overview and detailed design of OCSO. Section "Implementation" outlines OCSO prototype implementation, plus two optimization scenarios, i.e., Amazon EC2 [14] IaaS optimization and Amazon ElasticBeanstalk [15] PaaS optimization. Section "Experiments and evaluation" demonstrates and evaluates a series of experiments conducted over EC2 and ElasticBeanstalk. Section "Discussion" provides the discussion of OCSO approach. Finally, Section "Conclusions and future work" concludes the paper with summaries and future work.

Related work

In the context of cloud service efficiency optimization, much research focuses on optimizing service resource management through relevant resource scheduling/scaling techniques. Formal methods-based optimization approaches often rely on studying the relationships among the service's workload, deadline, cost, resource utilization, etc. With certain given constraints (e.g., time, budget, resource), the approaches mostly rely on a diversity of threshold controlled algorithms such as workload/deadline-constrained [16,17], and cost/budget-aware scaling management solutions [18,19]. While commonly resting on linear programming methods, they generally have limited applicability considering a diversity of user requirements, whereas they usually fail to function effectively while facing sophisticated unpredicted workloads.

On the other hand, heuristics-based resource management approaches usually count on relevant analytical models to facilitate optimized resource provisions and allocations. For instance, the power models with utilization threshold algorithm [6] are argued to assist dynamic VM placement and migration so that a considerable amount of energy consumption and CO2 emissions can be reduced compared with static resource allocation approaches. The cognitive trust model with dynamic levels scheduling algorithm [20] is proposed as a better resource scheduling technique which rests on resources trustworthiness and reliability parameters matchmaking. The resource allocation and application models with resource allocation and task scheduling algorithms [21] are advocated in which real-time task execution information is used to dynamically guide resource allocation actions. The workload prediction models with capacity allocation and load algorithms [22] is proposed to minimize overall VM allocation cost while meeting SLA requirements. The cost and time models with deferential evolution algorithms [23] would enable generating the optimal tasks schedules to minimize job completion cost and time. The Dual Scheduling of Cloud Services and Computing Resources models with Ranking Chaos algorithm [24] are designed to mitigate the inefficient service composition selection and computing resources allocation

issues. Additionally, IACRS [25] proposes a multi-metric group cloud-ready heuristic algorithm that would deal with compute, network and storage metric statistics simultaneously while performing scaling decisions. InterCloud [26] advocates an effective resource scaling approach seen as to distribute workload appropriately across multiple independent cloud data centers without compromising service quality of service (QoS) aspects.

Consequently, despite of their better resource scheduling and allocation outcomes, the primary objectives of all the above approaches either focus on minimized job completion time, meeting QoS/SLA requirements, or budget constraints, etc., whereas few of them can be implemented from the user end so that service users can achieve certain efficiency proactively. Moreover, none of them try to implement a native green resource utilization efficiency-oriented optimization, i.e., a typical approach through managing VM scaling up/down or in/out behavior regardless of varied workload dynamics.

Meantime, many industry cloud service providers allow monitoring service recourses through their native service interfaces, e.g., Amazon Web Services (AWS) has CloudWatch [27] and Rackspace uses Cloud Monitoring [28]. While most IaaS services offer VMs of a variety of sizes for users to select and scale from, the majority of PaaS service platforms are provisioned with automatic scaling capabilities so that the applications deployed over them can behave elastically despite of varied workloads, e.g., Amazon Auto Scaling [29], IBM SmartCloud Application policy-based automated scaling [30], Windows Azure Autoscaling Application Block (WASABi) [31], Rackspace Auto Scale [32]. Relying on similar threshold triggering algorithms which ask for a certain monitor metric for a specific interval/duration/breach time period, these policy/rule-based scaling solutions provide a simple and convenient means of customized resource scaling control depending on the applications' real-time monitor data. Specifically, they are designed to facilitate scaling in/out actions over certain numbers of equally sized VMs, where jobs (network traffics) are distributed evenly (mostly using a round-robin load balancing algorithm) through load balancers to each VM node.

However, for the reason that none of the above official industry service functions is primarily designed to achieve efficient resource utilization, these native service "add-ups" cannot facilitate ultimate green efficiency contributions. Specifically, for scaling up/down support, despite the monitor and alarm notification functions of Amazon CloudWatch and Rackspace Cloud Monitoring, while their VMs experience excessive high/low CPU utilization, the only available reactions are to shut down or terminate them and send notifications automatically (EC2 users can manually scale VMs up/down while they are in "stop" state) [14,33]. For scaling in/out operations, none of Auto Scaling (AWS), policy-based automated scaling (IBM), Auto Scale (Rackspace) or WASABi (Azure) tends to utilize "sophisticated" resource provision algorithms so that optimal numbers of VM can be provisioned instead of the "rough" scaling actions. Consequently, for such cloud VM resources that are not running efficiently, no solution would react appropriately to manipulate the scaling activities so that they can scale up/down while alone or in/out within a group towards the utmost each individual's green effectiveness.

In summary, existing resource scaling management approaches seldom directly address service resource utilization efficiency, whereas they have considerable limitations due to the inflexibility of using limited resource metric monitor as well as static threshold and scaling parameters. In contrast, OCSO is unique as a native and proactive service optimization approach that serves to achieve service resource utilization efficiency; OCSO's focus is the client-side green service optimization via dynamic scaling. Advanced from other existing approaches, OCSO's intelligent resource scaling algorithm utilizes dynamic threshold and scaling parameters under multiple service metric monitor statistics, which can instruct more accurate scaling actions considering the timing control as well as the scaling behaviors. In addition, OCSO is adaptable for implementation over multiple clouds, provided that there is a formal cloud service resource/interface/property description and orchestration specification framework available. One of the mainstream forces is OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [34]. Adding with such complete cloud service standard and reference support, OCSO has the potential to achieve much more for advanced service optimization scenarios.

OCSO system architecture

System overview

OCSO is an approach designed for service users to work with real-world cloud services to ensure efficient utilization of the provisioned service resources. OCSO system is rule-based, and once the mandatory service optimization parameters are completed, it would function fully automatically. As it detects inefficient service resource usage due to the workload changes, it reacts to add/remove certain resources by launching appropriate scaling up/down or in/out actions dynamically.

In our previous work [3], we have demonstrated the VM transition and workload reallocation approach that can actively transit VMs to appropriate VM sizes for varied workloads, called TARGO. In contrast, OCSO is a more advanced approach as: 1) it works for a wider range of cloud services, i.e., both IaaS and PaaS services; 2) it runs a more advanced and sophisticated threshold algorithm that rests on dynamic threshold and scaling

parameters. Figure 1 describes the architecture of OCSO including its five components, namely Optimization Gateway, Optimization Facilitator, Optimization Executor, Optimization Rule Repository, Logging and Notification Controller, as well as their data and process dependencies.

Optimization Gateway

Optimization Gateway is seen as the interface between OCSO system components and service provider clouds. It sends various service requests through OCSO cloud service APIs, which are developed using official cloud service API libraries. There are three types of service requests in general: general service information request for retrieving service specification, setting and status information; service utilization data request for acquiring service resource monitor data; and service manipulation request which makes certain changes to the services. These requests are launched by Service Information Collector, Optimization Facilitator and Optimization Executor respectively.

Optimization Facilitator

Optimization Facilitator comprises Service Resource Utilization Monitor and Utilization Data Regulator two components which work together handling service metric monitors and regulating resource utilization data. On detecting dramatic utilization changes which imply that the service instance is running inefficiently, it makes appropriate optimization decisions and then triggers appropriate optimization actions according to user specified optimization rules. Currently, OCSO supports both IaaS and PaaS scaling optimizations under a threshold triggering algorithm. Therefore, there are two separate sets of rule parameters. Shown in Figure 2, the algorithm takes input of optimization rules. Then, according to relevant monitoring period, frequency, up/down green limits and thresholds specified in the optimization rules, OCSO implements periodical service resource monitor actions. Then if the regulated monitoring utilization data violates the up/down green limits for the respected thresholds, appropriate scaling optimization will be triggered (refer to full algorithm in Additional file 1).

OCSO rests on enabling proactive and effective vertical scaling for IaaS services and horizontal scaling for PaaS services, which are controlled by a dynamically adjusted threshold triggering algorithm. Specifically, for IaaS optimization, when the (dynamic) up/down threshold

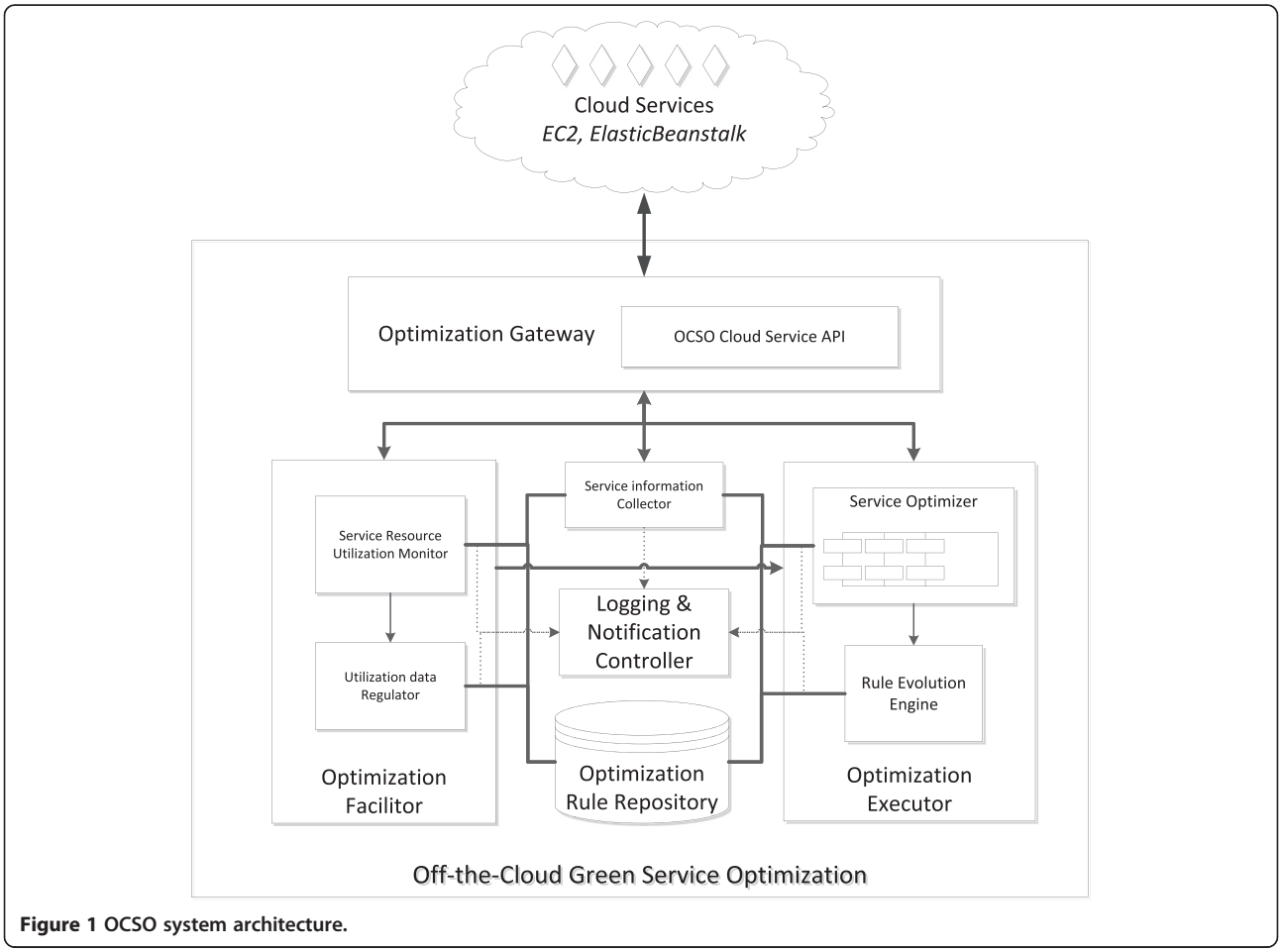


Figure 1 OCSO system architecture.

INPUT:
optimizationRules R_1, R_2, ..., R_n
1. **FOR** R_1 to R_n
2. INITIALIZE $schedule_n$, $timer_n$, $upCounter_n$, $downCounter_n$ $serviceUtilizationList$,$UtilizationRgulator$, Rule Evolution
3. GET $VMID_n$, $applicationName_n$, $serviceType_n$, $serviceProvider_n$, $period_n$, $frequency_n$, $metric_n$, $upLimit_n$, $downLimit_n$, $upThreshold_n$, $downThreshold_n$
4. **END FOR**
5. **FOR** R_1 to R_n
6. SET $timer_n$ with $schedule_n$ at $frequency_n$ to START (**REPEAT**)
7. SET $upCounter_n$ to 0; SET $downCounter_n$ to 0
8. INITIALIZE and GET $serviceUtilization_n$ by CALL $UtilizationRegulator.getLatestRegulatedMonitorData$ with $VMID_n$/$applicationName_n$, $serviceType_n$, $serviceProvider_n$, $period_n$, $frequency_n$, $metric_n$
9. ADD $serviceUtilization_n$ to $serviceUtilizationList_n$
10. **IF** number of $serviceUtilization_n$ in $serviceUtilizationList_n$ > $period_n$/$frequency_n$ **THEN**
11. REMOVE the first $serviceUtilization_n$ from $serviceUtilizationList_n$
12. **END IF**
13. **FOR** each $serviceUtilization_n$ in $serviceUtilizationList_n$
14. **IF** $serviceUtilization_n$ < $downLimit_n$ **THEN**
15. INCREMENT $downCounter_n$
16. **END IF**
17. **ELSE IF** $serviceUtilization_n$ > $upLimit_n$ **THEN**
18. INCREMENT $upCounter_n$
19. **END ELSE IF**
20. DISPLAY notification information
21. **END FOR**
22. **WHILE** $DownCounter_n$ >= $DownThreshold_n$ or $UpCounter_n$ >= $UpThreshold_n$ **DO**
23. DISPLAY notification information
24. SET $timer_n$ with $schedule_n$ to STOP (**UNTIL** STOP)
25. SET $upCounter_n$ to 0; SET $downCounter_n$ to 0
26. CALL $scalingOtimization.scale$ with $VMID_n$/$applicationName_n$, $serviceUtilizationList_n$
27. CALL $RuleEvolution.evolveRule$ with $serviceUtilizationList_n$ (refer to (2.1) – (5))
28. **IF** no error occurred during scaling and rule evolution processes **THEN**
29. DISPLAY notification information
30. SET $timer_n$ with $schedule_n$ at $frequency_n$ to START
31. **END IF**
32. **ELSE THEN**
33. DISPLAY notification information (Failed, retry next time)
34. SET $timer_n$ with $schedule_n$ at $frequency_n$ to START
35. **END ELSE**
36. **END WHILE**

Figure 2 The optimization triggering algorithm.

is met for a VM, which implies that it is under/over sized for the real-time workload, OCSO scales it up/down by transiting it and reallocating its workload to a successor VM of the green optimal size (considering its real-time workload volumes). For PaaS optimization, when the current application environment monitor data violates its current green up/down limit for the respected threshold, which indicates the environment is under/over provisioned, OCSO scales it in/out with the exact green optimal numbers of VMs (necessity for the real-time workload).

Optimization Rule Repository and optimization rule formats

Optimization Rule Repository stores optimization rules which detail relevant parameters for each optimization type. It instructs OCSO how to implement cloud service monitoring and optimizations for IaaS and PaaS respectively. While the monitor parameter and threshold parameters are generally the same, the optimization parameters vary significantly. Figure 3 illustrates a couple of rule layout examples representing the IaaS and PaaS optimizations supported by OCSO. Rules parameters are presented with standard XML syntax. A rule starts with "type" which tells the optimization type. Generally, a rule has three sections. The first section involves some general information such as the service information, rule name, id, times, etc. The second part details various details regarding the relevant optimization monitor, green boundary and trigger parameters. The last part comprises notification information.

Take the sample PaaS rule as an example, it provides OCSO the following instruction: I) This is an optimization designed for "PaaS" service of "Amazon ElasticBeanstalk". II) The rule is associated with an application named "2602test". III) Notification is switched "On" and data will be sent to the destination Email address. IV) The system periodically collects "Average" "CPUUtilization" data at the "Period" of "40" minutes and "Frequency" of every "5" minutes. V) With the specified green boundary between "40-80" in "Percentage", the respective "UpCounter" and "DownCounter" will be updated if the application monitor data violates the limits. VI) If either the "UpThreshold" of "5" or "DownThreshold" of "3" is met, a scaling optimization will be initiated. VII) Some secondary metric monitors are also implemented, seen as "RequestCount" and "NetworkIn" using the same monitor parameters as for the "Dominant" CPU metric.

Optimization Executor

Optimization Executor implements optimizations and updates the rules according to both the excessive amount of workload for the original service resource provision and the new provisioned resource capacity. For IaaS and PaaS optimizations there are different scaling control mechanisms which execute separate optimization processes (refer to the two scenarios in the next section).

Sample IaaS optimization rule

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Rule type="IaaSOptimization">
    <Provider>EC2.AWS</Provider>
    <RuleId>1363739083143</RuleId>
    <Time>Wed Mar 5 00:24:43 GMT 2014</Time>
    <VmId>i-4aesfe73</VmId>
  - <Optimization metric="CPUUtilization">
      <Statistics>Average</Statistics>
      <Unit>Percentage</Unit>
      <Period>20</Period>
      <Frequency>2</Frequency>
      <UpLimt>80</UpLimt>
      <DownLimit>30</DownLimit>
      <UpThreshold>8</UpThreshold>
      <DownThreshold>8</DownThreshold>
      <UpCounter>4</UpCounter>
      <DownCounter>1</DownCounter>
    </Optimization>
    <Notification>On</Notification>
  - <Notification method="Email">
      <Destination>d.fang@napier.ac.uk</Destination>
    </Notification>
  </Rule>
```

Sample PaaS optimization rule

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Rule type="PaaSOptimization">
    <Provider>ElasticBeanstalk.AWS</Provider>
    <RuleId>1422693735628</RuleId>
    <Time>Wed Mar 5 10:26:49 GMT 2014</Time>
    <ApplicationName>2602test</ApplicationName>
    <Notification>On</Notification>
  - <Optimization metric="CPUUtilization">
      <Statistics>Average</Statistics>
      <Unit>Percentage</Unit>
      <Period>40</Period>
      <Frequency>5</Frequency>
      <UpLimt>80</UpLimt>
      <DownLimit>40</DownLimit>
      <UpThreshold>5</UpThreshold>
      <DownThreshold>3</DownThreshold>
      <UpCounter>4</UpCounter>
      <DownCounter>1</DownCounter>
      <Dominant>Yes</Dominant>
    </Optimization>
  - <Optimization metric="RequestCount">
      <Statistics>Average</Statistics>
      <Period>40</Period>
      <Frequency>5</Frequency>
      <Dominant>No</Dominant>
    </Optimization>
  - <Optimization metric="NetworkIn">
      <Statistics>Average</Statistics>
      <Period>40</Period>
      <Frequency>5</Frequency>
      <Dominant>No</Dominant>
    </Optimization>
  - <Notification method="Email">
      <Destination>d.fang@napier.ac.uk</Destination>
    </Notification>
  </Rule>
```

Figure 3 Sample OCSO optimization rules.

Optimization rules are evolved to make sure the new rule can best instruct the next optimization. Basically, if the system detects that there is excessive high/low workload incoming than the provisioned resource could possibly handle, the next up/down green limit will be adjusted to relatively a lower/higher value whilst the threshold will probably also be tuned to a smaller value (depending on whether it is a continued scaling). As a consequence, the next optimization would be triggered more easily if the workload continues to increase/decrease. Then the updated values are validated with the new provisioned VM resources as well as against appropriate green boundary regulations.

Logging and Notification Controller

Due to the automatic and self-initiative nature of OCSO optimization, it is essential to inform users what has happened before, during and after the optimization and leave complete service optimization trace records. Logging and Notification Controller enters the resource information, utilization, decision and optimization details to logs and notifies the user where necessary and at each critical state.

Implementation

OCSO prototype is implemented in Java. Currently it is fully integrated with Amazon EC2 and ElasticBeanstalk using OCSO API originated from AWS Java SDK. As illustrated in Figure 4, it allows users to view their owned service instances and other information through its built-in service panes. For instance, for EC2, VMs' ID, status, size, etc. are displayed; for ElasticBeanstalk, applications' name, health, instances attached, etc. are showed. The buttons in the middle of the panels provide service optimization management: "Console" opens the console log subpanel of the system; "Monitor" opens the service utilization monitor subpanel; "New Rule", "Delete Rule" and "Modify Rule" allow users to create and edit optimization rules. To optimize a service, a user simply selects the target from the owned IaaS VMs or PaaS applications, and then enters a series of parameters, e.g., period(s), frequency, monitor metric(s), thresholds, green up/down limits, notification methods, etc. The example given in Figure 4 (the top "IaaS Optimization panel") illustrate how an optimization rule is created for the VM instance with ID "i-a1fcd8e0" with parameters of the following: period: "10" (minute), frequency "2" (times per minute), green up limit: "80" and down limit: "40" (% of CPU usage), up and down threshold: "4", "3" (times), and notification: "d.fang@napier.ac.uk" (by email). As the optimization rules are completed, the optimizations would begin once the user clicks the "Start/Restart Optimization" button. Afterwards, the optimizations would run on their own initiatives continuously, whereas it can be interrupted manually. OCSO does not need any human intervention during the optimization. In addition, it would log any critical optimization data and events when necessary whilst notification emails will be sent automatically if the user chooses to do so.

Scenario 1: EC2 IaaS optimization

IaaS optimization works similarly as TARGO: by periodically monitoring the CPU utilization of the target VM,

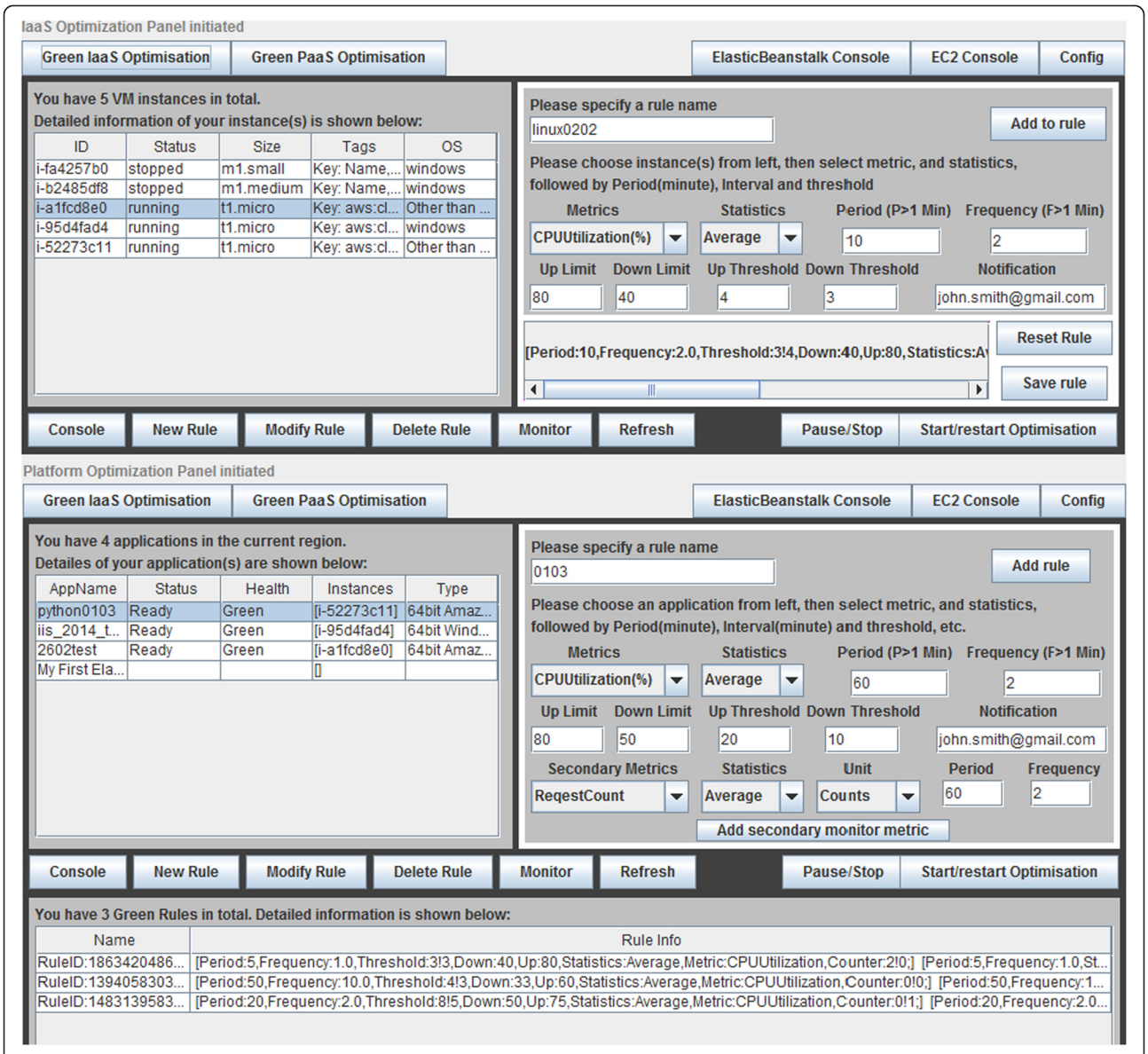
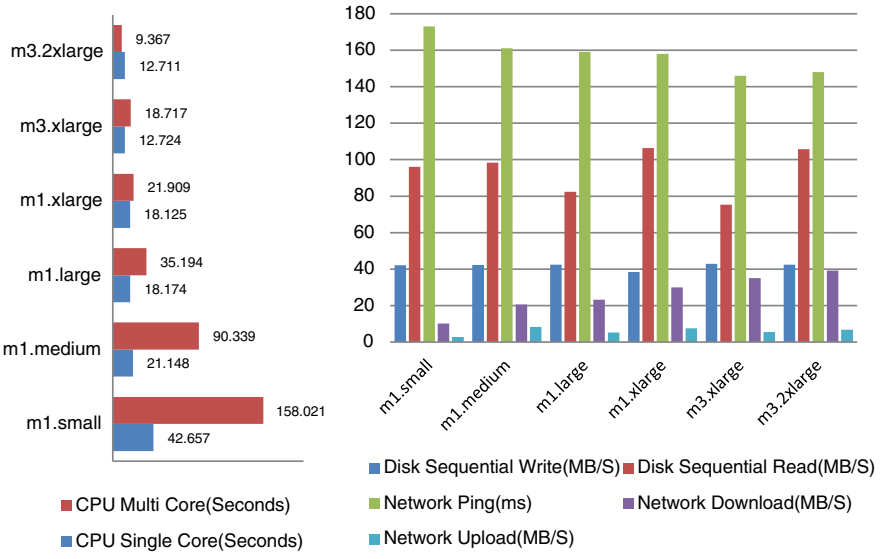


Figure 4 OCSO prototype screenshots.

the system reacts to scale it up/down proactively whenever the real-time workload changes concretely. The VM live scaling is performed by transiting the original VM to its successor in the optimal size that would fit the varied workload and then reallocate the workload to it. Therefore, it appears to the user that the optimized VM is operating at a dynamically adjusted size that can always run its workload green efficiently.

In order to guide such optimization, TARGO introduces VM performance gap ratios (PGRs). They are seen as the performance differences among VMs of distinct sizes (with the same VM image and other configuration settings). Figure 5 shows three series of VM performance statistics with regard to their capabilities of processing CPU-intensive tasks and throughput of disk and network performances. It can be seen from the data that, the larger the size a VM is, the quicker it completes the compute-intensive tests (considering both their single and multi-core performances) whilst the better the overall throughout is. Then, from the overall differences, a series of PGRs are formulated, which are used to guide appropriate scaling up/down actions from an inefficient VM size to the most efficient size. For instance, if a t1.micro VM runs a workload with 99.9% CPU utilization, an m1.small would manage it with a CPU usage of approximately 24% whilst an m1.medium would run at only 12%. Conversely, if a workload causes a VM of size m3.double extra large running at 71% CPU usage, it would drive an m3.extra large running at full load.



EC2 instance types	T1. Micro	M1. Small	M1. Medium	M1. Large	M1. Extra large	M3. Extra large	M3. Double extra large
Performance Gap Ratio:	0.24	0.51	0.57	0.69	0.61	0.71	

Figure 5 Performances and PGRs of Amazon EC2 VMs.

The known limitation of TARGO approach is the optimization lags. As a result, it cannot strictly hold the VMs in the specified green efficient boundary with its fairly simple threshold algorithms. In fact, this is due to the fact that threshold mechanisms would act only after green limit violations, by which time the VMs have already been running inefficiently for a while. This work eliminates such limitation by advocating OCSO IaaS optimization, which rest on optimized threshold algorithms and rule evolution equations as following:

For resource utilization regulation:

$$RV^t = \frac{F}{P} * \sum_{n=1}^{P/F} MV_n^{tn} \tag{1}$$

For optimization rule parameter evolution:

1) While scaling up:

$$NewLimit_U = \begin{cases} 2 * CL_U - \frac{1}{CT_U} * \sum_{n=1}^{CT_U} RV_n^{tn}; \left(continued\ scaling\ up\ while \frac{\bar{RV}}{CL_U} < 110\% \right) \\ CL_U * 90\%; \left(continued\ scaling\ up\ while \frac{\bar{RV}}{CL_U} > 110\% \right) \\ OL_U; (intermittent\ scaling\ up\ or\ no\ optimization\ needed) \end{cases} \tag{2.1}$$

$$NewLimit_D = \begin{cases} CL_U * Ratio_{M(O-N)}; \left(if\ OL_D > CL_U * Ratio_{M(O-N)} \right) \\ CL_U * Ratio_{M(O-N)}^2; \left(if\ OL_D < CL_U * Ratio_{M(O-N)}^2 \right) \\ OL_D; (else) \end{cases} \tag{2.2}$$

$$NewThreshold_U = \begin{cases} CT_U * 90\%; (continued\ scaling\ up) \\ OT_U; (intermittent\ scaling\ up\ or\ no\ optimization) \end{cases} \quad (2.3)$$

where $NewLimit_D >= 10; NewThreshold_U >= 1;$

2) While scaling down:

$$NewLimit_D = \begin{cases} \frac{1}{CT_D} * \left(\sum_{n=1}^{CT_D} RV_n^{tn}\right); \left(continued\ scaling\ down\ while \frac{\bar{RV}}{CL_U} > 90\%\right) \\ CL_D * 110\% \left(continued\ scaling\ down\ while \frac{\bar{RV}}{CL_U} < 90\%\right) \\ OL_D; (intermittent\ scaling\ down\ or\ no\ optimization\ needed) \end{cases} \quad (3.1)$$

$$NewLimit_U = \begin{cases} \frac{CL_D}{Ratio_{M(O-N)}}; \left(if\ OL_U < \frac{NL_D}{Ratio_{M(O-N)}}\right) \\ \frac{CL_D}{Ratio^2_{M(O-N)}}; \left(if\ OL_U > \frac{NL_D}{Ratio^2_{M(O-N)}}\right); \\ OL_U; (else) \end{cases} \quad (3.2)$$

$$NewThreshold_D = \begin{cases} CT_D * 90\%; (continued\ scaling\ down) \\ OT_D; (intermittent\ scaling\ down\ or\ no\ optimization) \end{cases} \quad (3.3)$$

where $NewLimit_U <= 90; NewThreshold_D >= 1;$

Here RV^t stands for the regulated value at specified frequency F in monitor period P for that moment of time t. MV_n^t indicates the nth raw monitor data value recorded at time tn. U means "Up" and D means "Down". CT is the current threshold value whilst CL stands for the current limit, whereas OT and OL are original threshold and limit respectively. RV_n^{tn} represents the regulated utilization value at time t_n, $Ratio_{M(O-N)}$ means the respected PGR of the metric from the original instance size to the new size.

The regulated utilization values are calculated from the raw CPU usage monitor values recorded at the specified frequency for the entered period, using (1). This provides relatively stable monitor values that can reflect the overall resource utilization changes of the monitor period, compared with the raw real-time monitor data. Then, in order to mitigate the optimization delays, OCSO IaaS optimization adopts a dynamic adjusted threshold algorithm that uses variable green boundary limits and violation thresholds. Basically, the amount of dynamic scaling adjustment is controlled according to two aspects: I) whether an optimization is a continued or intermittent scaling; II) how much the regulated utilization monitor values exceed the green limits (i.e., within, by or over 10%). In this way, whenever an optimization occurs, the successor's green boundary limit and threshold values are evolved, using (2.1), (2.3), (3.1) and (3.3). More specifically, depending on whether its successor could keep up with the workload for the incoming period of monitor cycle, two sets of triggering parameters are used: in case of a continuous optimization, the respected up/down limit is lowered/raised for a certain amount depending on the utilization data recorded during the current monitor period (using (2.1)/(3.1)), whereas the respected threshold value is lowered (using (2.3)/(3.3)); in case of intermittent optimization or no optimization, the evolved parameters are restored to the initial user specified values. Additionally, whenever an up/down limit is changed, the other limit must be validated (using the relevant PGR). This is to ensure that the gap between the up and down limits is always appropriate for later optimizations, regardless of the volumes of workload. Without this validation and the mandatory supplement updates, as the gap becomes

too large or too small, the optimizations would either miss the optimal timing or be triggered too early. The validation updates stay the same as they are in TARGO, using (2.2) and (3.2). The above complete rule evolution process creates a dynamic green boundary and respected thresholds specifically for every new VM successor considering its size as well as the current workload volume. In this way, the system would act more proactively while responding to extremely altered workloads. By doing so, OCSO IaaS optimization overcomes TARGO's lagging limitations by upsizing VMs quicker while facing dramatically/continuously increasing workloads and downsizing VMs earlier in case of rapidly/constantly descending workloads.

EC2 IaaS optimization flow

OCSO IaaS optimization flow is illustrated in Figure 6. Basically, the contents in the left column are executed by OCSO, whereas the right one shows the flow activities in EC2. The scaling optimization starts when a VM violates its green limit for the specified threshold. Firstly, the optimal VM size is calculated according to the period's resource utilization data against its provisioned VM size. Here, new rule parameters are also prepared for use of the VM successor once it is successfully deployed. Then, the system requests the detailed settings of the current VM from EC2 such as VM image information, resource settings, security setups, IP address, etc. on receiving the request, EC2 replies the information. Next, an EC2 VM instance creation request is initiated using the creation parameters gathered earlier, which is to be handled by EC2. While the request is being handled, the system periodically requests and checks the status of the new successor VM. Once EC2 notifies that the new VM is up and running, the profile information (IP address, name tag, etc.) of the original VM is transferred to its successor, and the workload is automatically passed to the new VM (due to the IP address reallocation). Accordingly, the optimization rule is updated for the successor. Finally, the original VM termination request is sent and executed by EC2.

To the service user, as the service is being optimized, every VM successor launched would be exactly the same as the previous one. This is seen as each one of them is of the same VM image, settings and profiles; the only difference is that it is deployed in the green optimal size for the real-time workload volume. Therefore, a successor ought to work efficiently until the workload changes again, by then another optimization would be triggered. In this way continuously, IaaS resource utilization efficiency is achieved, since users do not need to run a constant over-provisioned VM for just occasional large volumes of workloads, or vice versa.

Scenario 2: ElasticBeanstalk PaaS optimization

OCSO PaaS optimization implements optimized monitor and scaling control which can implement scaling activities more effectively than official dynamic scaling solutions. With the proposed monitor and scaling control algorithms, OCSO calculates the optimal number of VMs to provision and performs the modification by requesting to add/remove VMs at the best scaling timing. In order to do so, OCSO PaaS optimization disables the applications' native automatic scaling functions and modifies the environments by requesting service environment updates with API requests on-the-fly. In case of only one VM is presented in the environment, OCSO would continue to scale in by downsizing the single VM, provided that the workload continued to drop. Consequently, OCSO achieves PaaS service resource utilization efficiency since the effective scaling controls would save considerable VM hours while managing varied workloads.

Except the application environment CPU utilization being monitored as the main basis to trigger scaling optimizations, frequently, some secondary monitor metric data can present useful information to assist in calculating the appropriate capacity to provision. For instance, the number of VMs needed is often proportional to the increased count of the application visits/requests, network in/out volumes, disk write operations, etc. In another words, while an application is running, it would build up a dynamic ratio which is of these metric data to the number of VMs needed for it (for a certain preferable environment resource provision). Here we name it as "CapacityRatio". It is seen an average value of the ratios of the monitored secondary metric data values to the number of VMs in the environment (refer to (4)).

$$CapacityRatio = \frac{\sum_{i=1}^{N}\left(\frac{\sum_{n=1}^{P/F} SecondaryMetric_n}{P/F * Capacity_n}\right)_i}{N} \tag{4}$$

Where N is the number of total VMs at specified frequency F in monitor period P.

Next, the new environment resource capacity (optimal VM number) and rule evolution take the form (the utilization regulation equation stays the same as (1)):

$$Capacity_{new} = \frac{\sum_{N=1}^{Capacity_C}\left(\frac{\sum_{n=1}^{T} RV_n^{tn}}{T}\right)_N}{U + D} + \frac{SecondaryMetric_C}{2 * CapacityRatio} \tag{5}$$

Here RV^t stands for the regulated value at time t, whereas T is the total number of collected utilization values (most likely the respect threshold value). U and D

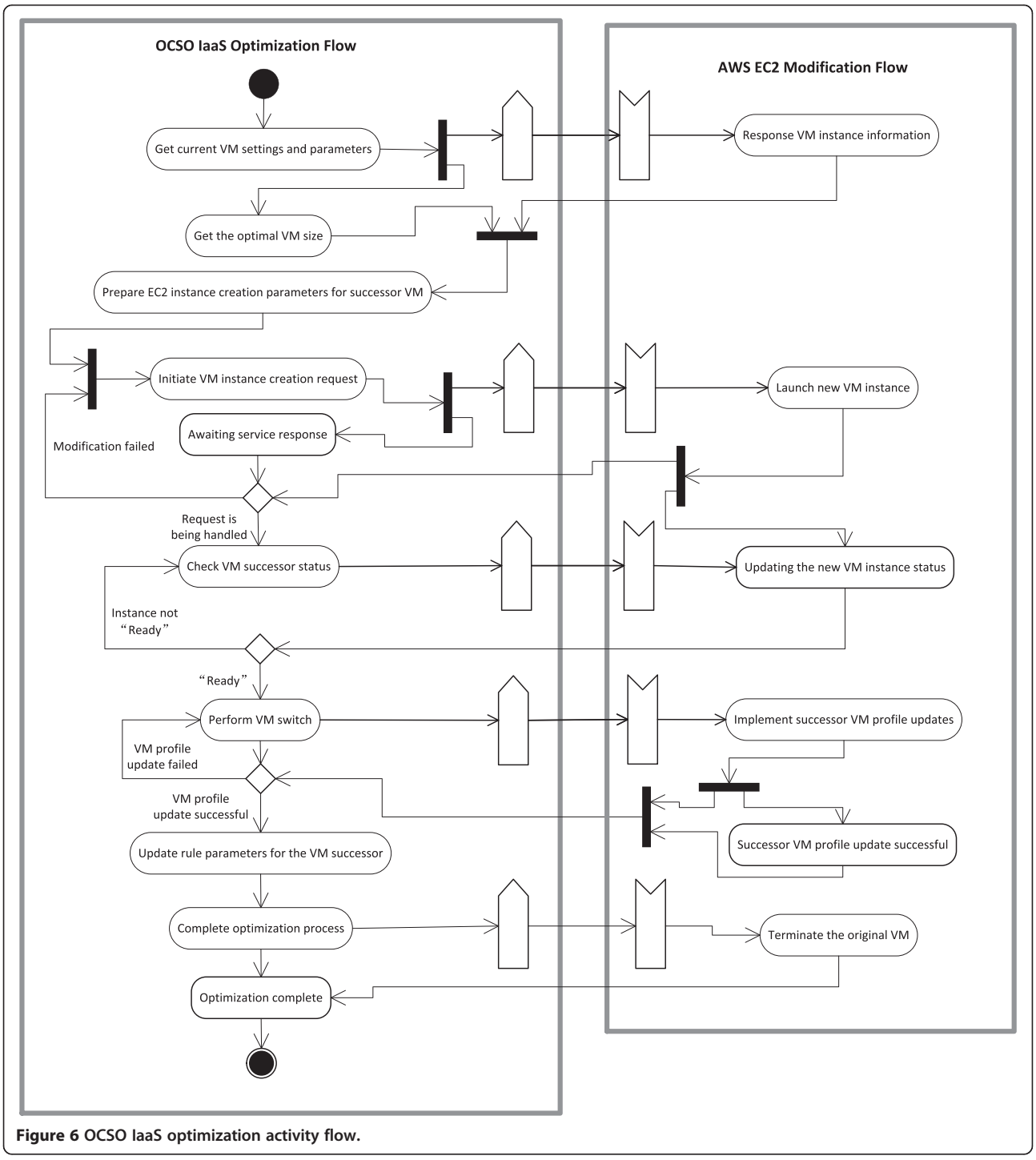


Figure 6 OCSO IaaS optimization activity flow.

means the up and down limit respectively. $Capacity_C$ represents the current capacity, i.e., the current total number of VMs in the environment.

The new application environment capacity is generated by considering two factors: the secondary metric data-reflected capacity; the current combined CPU utilizations of all involved VMs towards the specified green limit. They produce two provisional capacity values: the first is extracted by dividing the sum of all VMs' regulated monitor usage values by the optimal usage value (the average value of the green up and down limits); the second is produced by dividing the latest secondary metric monitor value by the CapacityRatio. Using (5), the final identical capacity values (number of VMs needed) is determined, which is the average value of the two provisional values. Additionally, OCSO PaaS optimization utilizes similar

dynamic green limit and threshold evolution as for its IaaS optimization using (2.1), (2.3), (3.1), (3.3), except that there is a gap of 30 (at least) between the new up and down limit, which is to prevent fault scaling.

ElasticBeanstalk PaaS optimization flow

OCSO PaaS optimization is initiated when the real-time green up/down threshold is met for an application. Figure 7 illustrates the flow diagram of OCSO PaaS optimization and the activities incurred in ElasticBeanstalk. Firstly, the application environment resource information is acquired from ElasticBeanstalk. With the responded application running parameters along with the calculated optimal VM numbers (environment capacity), the system initiates service modification request to update the capacity of the application environment. As the modification request is received, ElasticBeanstalk begins to handle the modification by launching appropriate number of VMs of the appropriate VM image and configuration, and then adding/removing the VMs from the load balancer (i.e. Elastic Load Balancing) which is attached with the application (These background service environment modifications are processed automatically by AWS). During this period, the application status becomes "updating". While the update is successfully implemented, OCSO verifies the modification against the updated application environment. Finally, the optimization rule parameters are updated for the new environment resource provision, which ends the optimization cycle.

Experiments and evaluations

Overview of the experiments

A series of experiments are conducted to evaluate the effectiveness of the proposed approach. The IaaS service used is Amazon EC2 whilst the PaaS service used is Amazon ElasticBeanstalk. The reasons for adopting AWS are: the overall EC2 VMs boot time is faster compared to other providers' such as Rackspace's [35] and GoGrid's

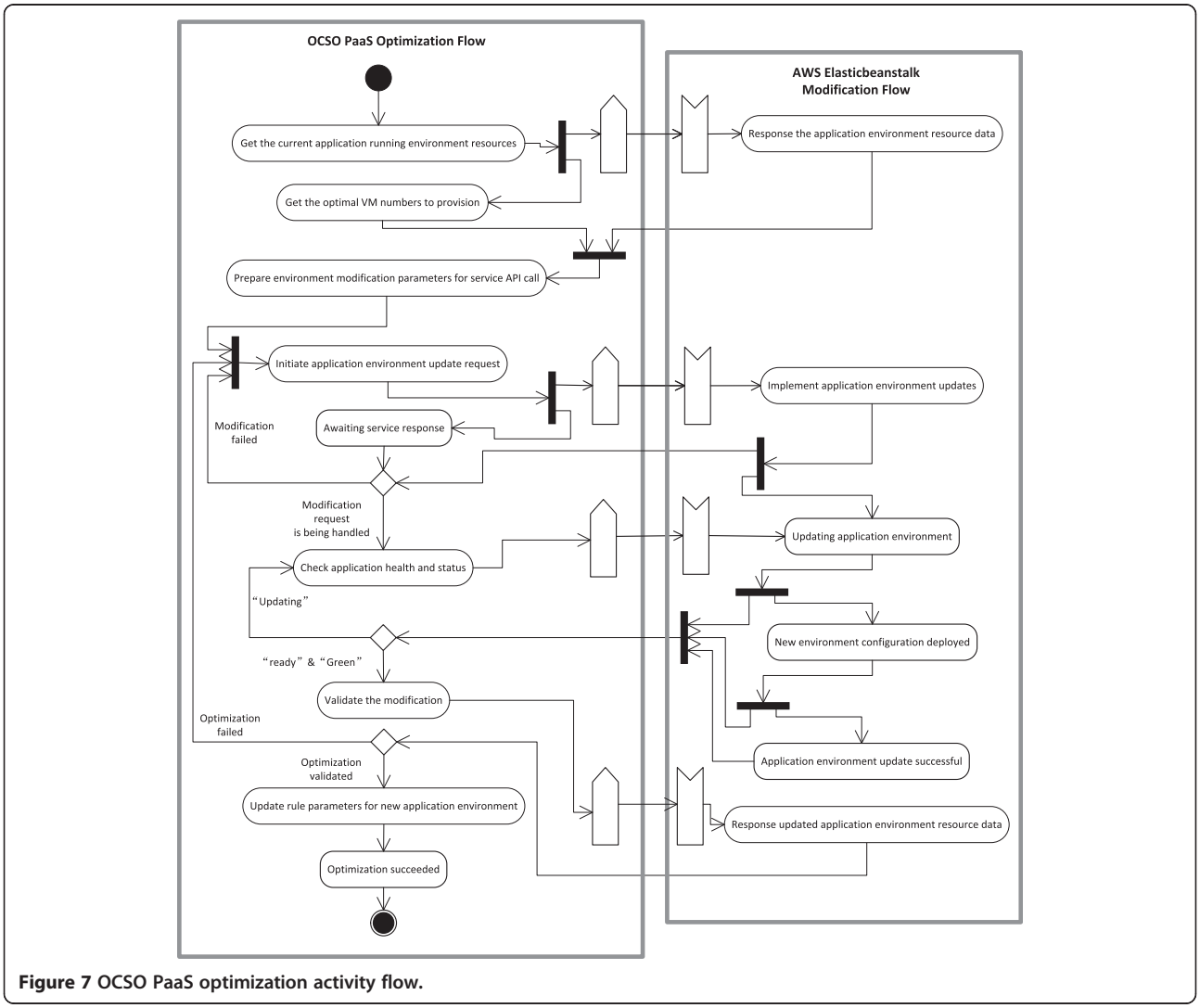


Figure 7 OCSO PaaS optimization activity flow.

[36]; ElasticBeanstalk offers more detailed control to the service components and their configurations (scaling options, monitor parameters, load balancers, VM groups, etc.) via service API in contrast with Google AppEngine [37] and WASABi [31]. For workload generation, Apache JMeter [38] is used to send scheduled workload in controlled volumes.

For both series of experiments, the allocated workloads are of different dynamics: at certain time intervals, they either alter slightly or dramatically. This is to test the effectiveness of OCSO when dealing with different workload patterns. The IaaS optimization experiment illustrates how OCSO can achieve improved result over TARGO as well as using EC2 VMs of static provision sizes, under the same workload dynamics. The PaaS optimization experiment demonstrates the differences between how ElasticBeanstalk and OCSO PaaS would scale while experiencing the same altered workloads.

EC2 IaaS optimization experiment and evaluation

Figure 8 illustrates the results of the experiments conducted over EC2 the EU region. The workloads used in the experiment have an overall span of 150 times of the initial unit value. For the first hour, it increases slowly at first and fairly dramatically until reaching the maximum value; subsequently, it decreases rapidly for the following 10 minutes and rather gradually until the end. Under such a workload, CPU utilizations of two ordinary VMs respond reasonably as the overall trend of the workload. When the workload goes beyond its capacity, the m1.xlarge VM becomes unresponsive, with its CPU utilization at constant 100% for nearly 20 minutes. The m3.2xlarge VM can load the work perfectly, yet the overall utilization is considerably low, with the majority of the usages under 50% throughout the experiment. This suggests that ordinary EC2 VMs in fixed sizes can hardly achieve optimal resource utilization efficiency in case of dynamic workload.

On the other hand, TARGO and OCSO IaaS optimization strive to retain the most suitable VM to meet resource utilization efficiency requirement fully automatically. The arrows in the figures designate the scaling up/down optimizations occurred from an instance to its successor. Seen in the two bottom CPU usage figures in Figure 8, with a green CPU utilization range of 40-80% and starting with VMs at size of m1.small, both approaches manages to scale the VMs up and down as the workload changes, by launching successors in m1.medium, and then m1.large... and finally ending at m1.

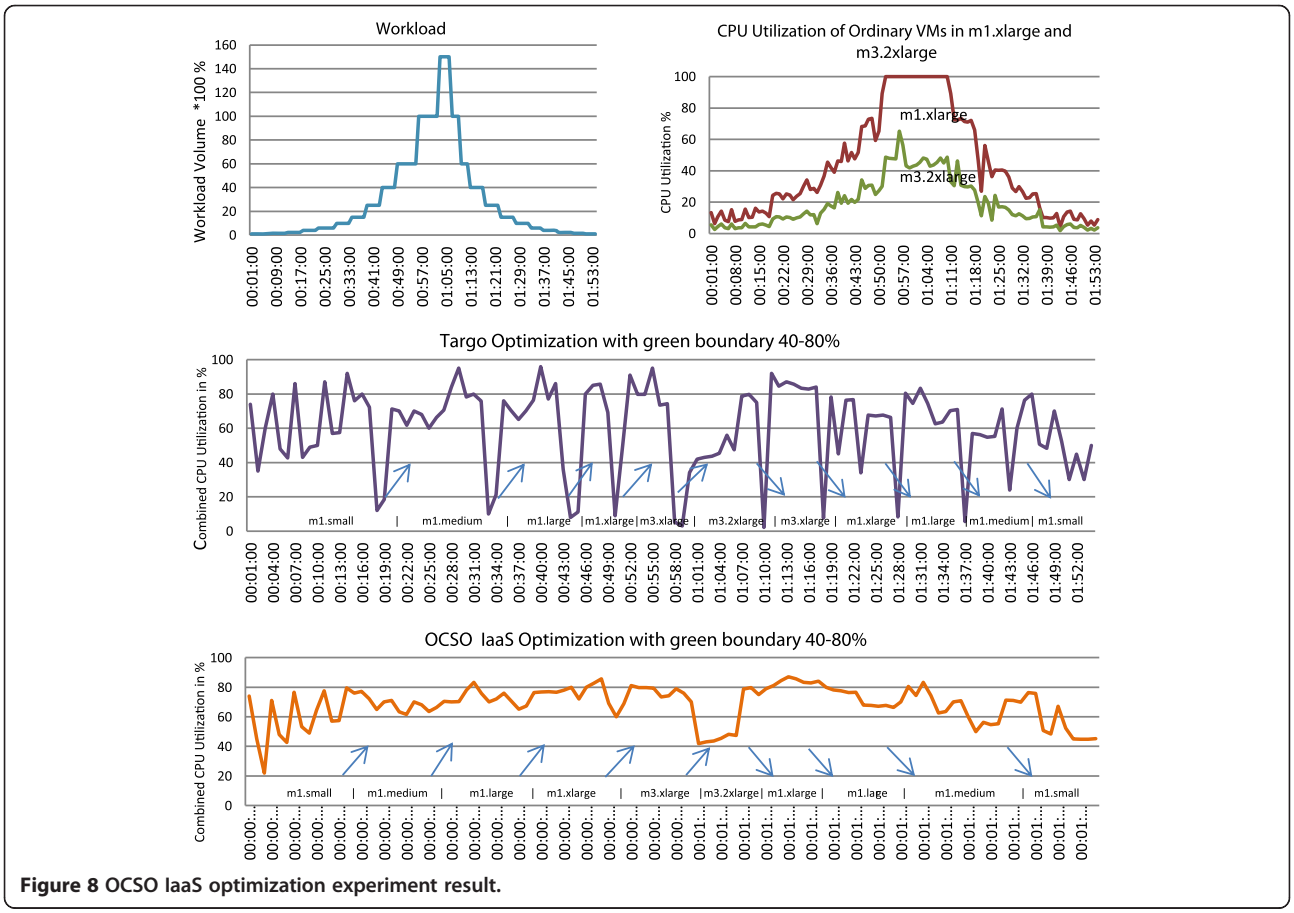


Figure 8 OCSO IaaS optimization experiment result.

small. Between these better results due to the scaling optimizations, it can be found that TARGO cannot guarantee the VM CPU utilizations to perfectly stay within the specified boundary, with plenty of usage values over 80% and less than 40% for the majority of the VMs. In contrast, OCSO IaaS optimization achieves more effective scaling optimizations by maintaining relatively stable VM CPU utilizations for most of the VMs. Due to the dynamic threshold and green limit evolution used in the optimized scaling algorithm, OCSO initiates faster optimization cycles to closely cope with the changes of the workload.

This series of experiments shows that the proposed IaaS optimization achieves distinguished outcome compared with ordinary EC2 VMs whilst it overcomes the limitations of TARGO. It can scale the original VM up/down as the workload varies, regardless of it is in a slow or fast pace. With a user specified green boundary, it can react instantly while the current VM is about to reach the up/down limit. Consequently, the VM resource utilization efficiency can be improved significantly with OCSO IaaS optimization.

PaaS optimization experiments and evaluation

A couple of experiments are conducted on ElasticBeanstalk the US region to evaluate the effectiveness of the proposed PaaS optimization approach against the native Amazon Auto Scaling functions (Figure 9). The sample worker tier application is deployed in Tomcat 7 which is contained in Amazon Linux 64-bit VMs. The scheduled workload requests in different volumes are sent to the application using Apache JMeter. With the illustrated patterned workload, we demonstrate the differences (with equivalent scaling settings) between using Auto Scaling and OCSO PaaS optimization (to scale freely from 1–20 m1.medium VMs with period of 5, breach/threshold of 4, monitor frequency of 1 and CPU usage up/down limits at 30-70%).

As Figure 9 demonstrates, the workload for the experiment can be divided into two parts: from start to

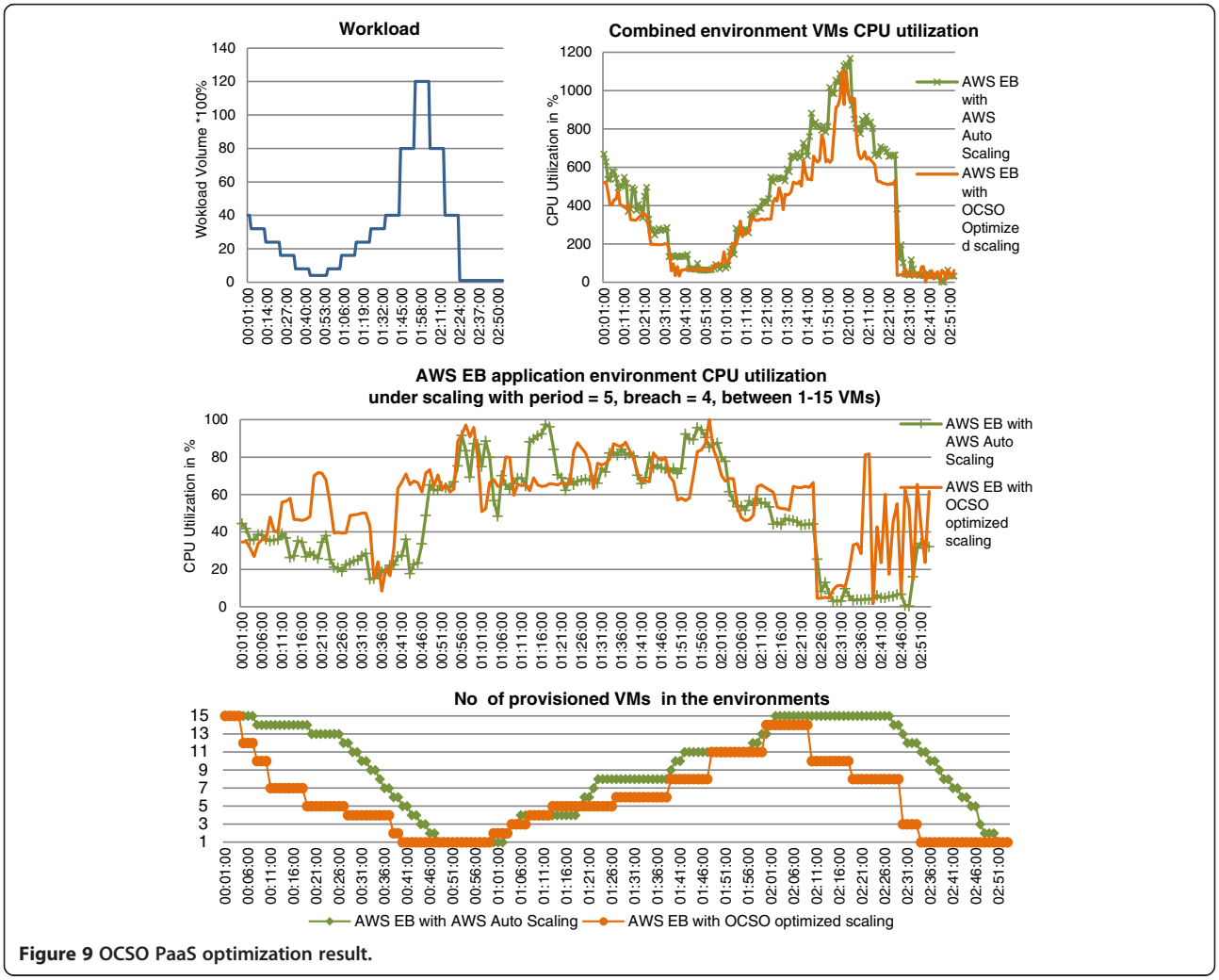


Figure 9 OCSO PaaS optimization result.

01:40:00, the workload decreases and increases gradually in a slow pace; afterwards, the increase and decrease become radically. Under such workloads, both scaling options manage to scale up and down as workload increases/decreases, seen as that both combinations of the VM CPU utilizations can reflect the general trend of the workload (refer to the "Combined environment VMs CPU utilizations" figure). Here the combined CPU utilization under OCSO optimization is slightly lower than it from using AutoScaling. The reason for this difference is that, OCSO utilizes smaller numbers of VMs for considerable amount of time whilst smaller numbers of VMs would mean smaller amount of VM (idle) running overhead, which then produces a smaller total CPU usages. Additionally, from the "AWS EB environment CPU utilizations" figure, the one under OCSO PaaS optimization shows relatively higher values than the other under AutoScaling. While the workload continuous descending and rising in such a rapid rate, both environment CPU utilizations could only be maintained within a rough boundary of 20-90%. Yet overall speaking, OCSO achieves a better result than Auto Scaling, seen as it owns more utilizations within the specified 30-70% green limit than AutoScaling, especially while scaling down. Moreover, as the workload drops to almost none, AutoScaling could only maintain a single original m1.medium VM despite of very low CPU utilizations. Nonetheless, OCSO is able to scale down to a m1.small VM as the last step to guarantee ultimate resource utilization efficiency (which is why the CPU utilization fluctuates and seems higher from 02:30:00).

The most obvious differences are that OCSO manages to provision smaller numbers of VMs for the workload for the majority time during the experiment, seen from the "No of provisioned VMs" figure in Figure 9. As the workload begins to fall, OCSO scales in much quicker than AutoScaling: the numbers of provisioned VMs are constantly lower than them from AutoScaling; it reaches the minimum VM number 10 minutes earlier than it. Then as the workload starts to climb, it scales out a little slower than AutoScaling. By the time the workload reaches its peak, AutoScaling scaled to 15 VMs in total whilst OCSO utilizes 14 VMs only. Subsequently, when the workload drops again and in a much faster pace, OCSO manages to scale in reasonably, whereas AutoScaling experiences considerable scaling. By the time AutoScaling scales to 1 VM, it is 15 minutes later compared with OCSO.

From the compared experiment results, it is noticeable that AutoScaling solution cannot scale in effectively while the volume of the workload continuous descending (regardless of gradually or dramatically), with its slow "one VM by another" manner. In contrast, OCSO can scales in much faster, whereas each optimization would reduce a group of VMs in an appropriate number according to the real-time utilizations. On the other hand, for scaling out effectiveness as workload increase, there are not many differences between the two approaches, except that OCSO scales out a little slower than AutoScaling. As a result, the overall outcome is that OCSO PaaS optimization utilizes far less necessary VMs (hours) while managing scaling actions to cope with the same workload dynamics. This suggests it a better solution considering the VM resource utilization efficiency. OCSO can achieve a better result due to the dynamic threshold and green up/down limit used in its intelligent scaling algorithm, whereas AutoScaling would fail to achieve ultimate VM utilization efficiency with its fairly simple auto scaling algorithm.

Discussions

As a unique approach designed for off-the-cloud use and by service users, OCSO is capable of achieving distinguished effectiveness in optimizing service resource consumption from a new perspective. Meanwhile, it still suffers from a series of concerns, which would be improved via introducing additional optimization modules. In fact, the optimization premise of OCSO implementation relies on the sufficiency and effectiveness of the target cloud services' resource monitor and configuration options. More specifically, they are reflected by the following service API and property factors.

Firstly, with regard to the various service interfaces which OCSO cloud service API rests on, the availability, stability and functionality of these official released service APIs are vital as they would determine the ultimate capability of OCSO approach. Fortunately, as seen from the current trend, all mainstream cloud service providers tend to provide the official versions of complete SDK/API kit/libraries for different types of users/developers for their mainstream services. Moreover, a series of third party cloud service APIs are becoming more and more mature in recent two years, such as Apache jclouds [39] and mOSAIC [40], this better supports OCSO as they offer an alternative route and sometime may even enable more advanced service manipulations. By seeing that both the official and the third-party service APIs have been maintained and updated regularly and "lively", we envision that OCSO approach would not be restricted by service API-related issues in the future. Ultimately, OCSO service API may later on become another service access and management alternative that allows service users to manipulate cloud resources effortlessly and effectively, as a unique interface.

Secondly, the effectiveness of OCSO is directly affected by a cloud services' functional and non-functional properties, such as options available for various service controls, elasticity and QoS attributes, and SLAs and security aspects. In case that the target cloud services (of the same kind) from different providers share similar

properties, there should not be many clear differences while optimizing them; however, if the target services share very different service properties, the effectiveness may vary distinctively. For IaaS optimization, the time for VMs creation would be a critical factor. For some providers, the long waiting time would compromise the overall optimization effectiveness, since OCSO would not be able to react as quickly as it should. This makes such service providers unideal while dealing with rapidly changing workloads (with/without the optimization). On the other hand, for PaaS optimization, certain service providers offer unique VM provision and scaling control options, e.g., the dynamic (frontend) VMs provisioned in Google AppEngine environment are "inaccessible" as an individual complete VM, but they can be added in as quickly as seconds; AppEngine users only needs to select min/max idle instances and pending latency for scaling parameters [37]. While dealing with the uniqueness, OCSO would need certain small modifications. As a matter of fact, through different optimization modules, specific off-the-cloud service optimizations can always be implemented for such providers, towards the best individual optimization outcome.

Conclusions and future work

Differently from the majority of efforts which are made in achieving energy-efficient service provision from the service provider perspective, in this paper, we present a novel user-side off-the-cloud service optimization solution that facilitates efficient service (resource) utilizations, knowingly OCSO. Typical formal methods or heuristics-based resource management optimizations as well as the official service provider resource scaling options expose various limitations in satisfying native green efficiency requirements when dealing with different workload patterns/types and multiple metric monitor data. To correct these weaknesses, an intelligent resource scaling algorithm is proposed for OCSO where the green boundary limits and thresholds are adjusted dynamically according to the real-time service resource utilization statistics. With the developed OCSO tool prototype, a number of experiments are conducted over Amazon EC2 and ElasticBeanstalk. The results prove that OCSO is uniquely able to: proactively scale the inefficiently running IaaS VMs up/down by transiting them to their successors at green optimal VM sizes and then reallocate the workloads to the successors automatically; effectively scale application environment resources in/out so that only necessary numbers of VMs are provisioned depending on the real-time workload volumes. To this extent, OCSO saves IaaS service resources by allowing users to use only necessary sized VMs and reduces overall numbers of VMs needed for applications deployed using PaaS service resources.

Although currently OCSO has limited service provider support, it is clear and promising that the proposed approach and algorithm can adapt and be deployed to a wide range of IaaS and PaaS services easily for very few modifications. The only limit of OCSO would be whether there are sufficient authorization and options available for adequate service access, request and configuration. In the future work, we will focus on providing more provider support for extended use cases and achieving service consumption efficiency through service compositions.

Additional file

Additional file 1: OCSO dynamic threshold triggering algorithm.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

DF form Edinburgh Napier University developed the algorithm and tool prototype. XL from Edinburgh Napier University supervised, tested the developments. LL from Tsinghua University and HY from Bath Spa University provided domain specific expertise in the evaluation of the approach. All Authors read and approved the final manuscript.

Acknowledgment

The work in this article has been sponsored by the Lawrence Ho Research Fund (LH-Napier2012).

Author details

[1]School of Computing, Edinburgh Napier University, 10 Colinton Road, Edinburgh EH10 5DT, UK. [2]School of Software, Tsinghua University, Beijing 100084, PR China. [3]Center for Creative Clusters, Bath Spa University, Newton Park, Newton St Loe, Bath BA2 9BN, UK.

Received: 14 March 2014 Accepted: 22 May 2014
Published online: 14 June 2014

References

1. Uddin M, Rahman AA (2012) Energy efficiency and low carbon enabler green IT framework for data centers considering green metrics. Renew Sust Energ Rev 16(6):4078–4094
2. Li C, Li Y (2012) Optimal resource provisioning for cloud computing environment. J Supercomput 62(2):989–1022
3. Fang D, Liu X, Liu L, Yang H (2003) TARGO: Transition and Reallocation Based Green Optimization for Cloud VMs. In: IEEE International Conference on Green Computing and Communications (GreenCom)., pp 215–223
4. Sousa L, Leite J, Loques O (2013) Green data centers: Using hierarchies for scalable energy efficiency in large web clusters. Inf Process Lett 113(14–16):507–515
5. Peoples C, Parr G, McClean S, Scotney B, Morrow P (2013) Performance evaluation of green data centre management supporting sustainable growth of the internet of things. Simul Model Pract Theory 34:221–242
6. Beloglazov A, Abawajy J, Buyya R (2012) Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. Futur Gener Comput Syst 28(5):755–768
7. Goiri Í, Berral LJ, Oriol Fitó J, Julià F, Nou R, Guitart J, Gavaldà R, Torres J (2012) Energy-efficient and multifaceted resource management for profit-driven virtualized data centers. Futur Gener Comput Syst 28(5):718–731
8. Jeyarani R, Nagaveni N, Srinivasan S, Ishwarya C (2013) ISim: A novel power aware discrete event simulation framework for dynamic workload consolidation and scheduling in infrastructure Clouds. Advances Intelligent Systems Computing 177:375–384
9. Etinski M, Corbalan J, Labarta J, Valero M (2012) Understanding the future of energy-performance trade-off via DVFS in HPC environments. J Parallel Distr Com 72(4):579–590

10. Huang C, Guan C, Chen H, Wang Y, Chang S, Li C, Weng C (2013) An adaptive resource management scheme in cloud computing. Eng Appl Artif Intell 26(1):382–389
11. Naumann S, Dick M, Kern E, Johann T (2011) The GREENSOFT Model: A reference model for green and sustainable software and its engineering. Sustainable Computing: Informatics and Systems 1(4):294–304
12. Moreno-Vozmediano R, Montero RS, Llorente IM (2011) Key challenges in Cloud Computing: Enabling the future internet of services. IEEE Internet Computing 17(4):18–25
13. Jansen W, Grance T (2011) Guidelines on Security and Privacy in Public Cloud Computing, NIST Special Publication 800–144. http://csrc.nist.gov/publications/nistpubs/800-144/SP800-144.pdf. Accessed 06 Oct 2013
14. Amazon Elastic Compute Cloud API Reference., http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-api.pdf. Accessed 05 Sep 2013
15. Amazon Elastic Beanstalk Developer Guide. http://s3.amazonaws.com/awsdocs/ElasticBeanstalk/latest/awseb-dg.pdf. Accessed 05 Sep 2013
16. Abrishami S, Naghibzadeh M, Epema D (2013) Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds. Futur Gener Comput Syst 29(1):158–169
17. Bossche RV, Vanmechelen K, Broeckhove J (2013) Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. Futur Gener Comput Syst 29(4):973–985
18. Mao M, Li J, Humphrey M (2010) Cloud Auto-Scaling with Deadline and Budget Constraints, 11th IEEE/ACM International Conference. pp 41–48
19. Mao M, Humphrey M (2013) Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows, IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)., pp 67–78
20. Wang W, Zeng G, Tang D, Yao J (2012) Cloud-DLS: Dynamic trusted scheduling for Cloud computing. Expert Syst Appl 39(3):2321–2329
21. Li J, Qiu M, Ming Z, Quan G, Qin X, Gu Z (2012) Online optimization for scheduling preemptable tasks on IaaS cloud systems. J Parallel Distr Com 72(5):666–677
22. Ardagna D, Casolari S, Colajanni M, Panicucci B (2012) Dual time-scale distributed capacity allocation and load redirect algorithms for cloud systems. J Parallel Distr Com 72(6):796–808
23. Tsai J, Fang J, Chou J (2013) Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm. Comput Oper Res 40(12):3045–3055
24. Laili Y, Tao F, Zhang L, Cheng Y, Luo Y, Sarker B (2013) A Ranking Chaos Algorithm for dual scheduling of cloud service and computing resource in private cloud. Comput Ind 64(4):448–463
25. Hasan MZ, Magana E, Clemm A, Tucker L, Gudreddi SLD (2012) Integrated and Autonomic Cloud Resource Scaling. In: Integrated and Autonomic Cloud Resource Scaling, IEEE Network Operations and Management Symposium (NOMS)., pp 1327–1334
26. Calheiros R, Toosi A, Vecchiola C, Buyya R (2012) A coordinator for scaling elastic applications across multiple clouds. Futur Gener Comput Syst 28 (8):1350–1362
27. Amazon CloudWatch Developer Guide. http://awsdocs.s3.amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf. Accessed 07 Sep 2013
28. Rackspace Cloud Monitoring Developer Guide. http://docs.rackspace.com/cm/api/v1.0/cm-devguide/cm-devguide-20140530.pdf. Accessed 01 Jan 2014
29. Auto Scaling Developer Guide., http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-dg.pdf. Accessed 10 Oct 2013
30. Policy-Based Automated Scaling for IBM SmarCloud Application Services. http://www.ibm.com/cloud-computing/uk/en/paas.html. Accessed 23 Dec 2013
31. Autoscaling Application Block. http://azure.microsoft.com/en-us/documentation/articles/cloud-services-dotnet-autoscaling-application-block/. Accessed 15 Oct 2013
32. Auto Scale Developer Guide. http://docs.rackspace.com/cas/api/v1.0/autoscale-devguide/autoscale-devguide-20140612.pdf. Accessed 18 Oct 2013
33. Ferraris FL, Franceschelli D, Gioiosa MP, Lucia D, Ardagna D, Di Nitto E, Sharif T (2012) Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds, 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp 423–429
34. TOSCA Overview. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview. Accessed 10 May 2014
35. Rackspace Next Generation Cloud Servers Developer Guide. http://docs.rackspace.com/servers/api/v2/cs-devguide/cs-devguide-20140529.pdf. Accessed 19 Jan 2014
36. GoGrid Cloud Server User Manual. https://wiki.gogrid.com/index.php/Cloud_Server_User_Manual. Accessed 09 Dec 2013
37. Google AppEngine. https://developers.google.com/appengine/. Accessed 02 Jan 2014
38. Apache JMeter. https://jmeter.apache.org/index.html. Accessed 03 Aug 2013
39. Apache jclouds. http://jclouds.apache.org/. Accessed 04 Apr 2014
40. Petcu D, Macariu G, Panica S, Crăciun C (2013) Portable Cloud applications—From theory to practice. Futur Gener Comput Syst 29(6):1417–1430

doi:10.1186/s13677-014-0009-1
Cite this article as: Fang *et al.*: **OCSO: Off-the-cloud service optimization for green efficient service resource utilization.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:9.