Profiling-as-a-Service: Adaptive Scalable Resource Profiling for the Cloud in the Cloud

Nima Kaviani[1], Eric Wohlstadter[1], and Rodger Lea[2]

[1] Department of Computer Science
[2] Department of Electrical and Computer Engineering,
University of British Columbia
201-2366 Main Mall, Vancouver, B.C. V6T 1Z4 Canada
{nkaviani,wohlstad}@cs.ubc.ca, rodgerl@ece.ubc.ca

Abstract. Runtime profiling of Web-based applications and services is an effective method to aid in the provisioning of required resources, for monitoring service-level objectives, and for detecting implementation defects. Unfortunately, it is difficult to obtain accurate profile data on live client workloads due to the high overhead of instrumentation. This paper describes a cloud-based profiling service for managing the tradeoffs between: (i) profiling accuracy, (ii) performance overhead, and (iii) costs incurred for cloud computing platform usage. We validate our cloud-based profiling service by applying it to an open-source e-commerce Web application.

Keywords: Cloud Computing, Resource Monitoring, Application Profiling.

1 Introduction

Dynamic runtime instrumentation of applications is an effective method for understanding application behavior, but imposes significant overhead to the overall execution of the application [2,7,8,10]. One approach to mitigating this overhead is offline profiling which allows the profiling process to be executed in a controlled environment, using collected traces from a previously running application. However, relying on offline collected traces often leads to inaccurate or incomplete datasets which may not represent the full spectrum of application execution states [9].

With the emergence of cloud computing and its direct mapping of resource usage to financial costs, the need to understand the low-level behavior of services and applications has become more critical, yet the challenges in profiling stay the same. However, cloud computing offers unique features which we believe can mitigate some of the above concerns. In particular, elastic and adaptive resource usage can be utilized to provide realtime analysis of system behavior with minimal performance degradation. This is achieved essentially by selectively and adaptively instrumenting only a specific subset of virtual machine (VM) instances of a deployed application.

This approach, which we refer to as *Profiling-as-a-Service* (PraaS), offers adaptive instrumentation strategies that can collect realistic profiling information about running applications in the cloud while respecting desired quality of service requirements (QoS) (e.g., response time, throughput, and cost of deployment). Such QoS strategies need

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 157–171, 2011.
© Springer-Verlag Berlin Heidelberg 2011

to adhere to both *business* and *performance* requirements specified for an application undergoing instrumentation and monitoring. Essentially, a profiling service should help to manage tradeoffs between three factors:

- *Accuracy:* Accurate profiling information is important for software developers who need to make decisions using this data, under tight business schedules. Unfortunately, accuracy could come at the expense of performance or financial costs. Collecting detailed profiling information results in application performance degradation. Performance degradations often are tried to be overcome by supplying more resources (CPU, memory, etc.) which in turn imply higher execution costs for the application under instrumentation.
- *Performance:* Cloud-based services and applications must ensure high performance to meet expected service-level agreements and good user experience. In the cloud, performance can be obtained through elastic scaling of virtual machine (VM) instances. Unfortunately, a naive approach to scaling could be wasteful and require unnecessary financial cost.
- *Cost:* Public cloud providers offer flexible infrastructure for system scaling and reconfiguration but obviously “there is no free lunch”. A profiling service will need to consider the financial costs of ensuring good accuracy and performance.

PraaS allows system architects to define policies describing their desired level of accuracy for collecting profiling information, expected QoS, and cost constraints. These policies are then uploaded to a PraaS cloud service, together with the original code for the target application, where the application is instrumented and deployed for resource usage monitoring. During application execution, PraaS will accommodate the application with just enough resources from the cloud to satisfy the specified constraints. We present our implementation of PraaS and evaluate it for a stateless, horizontally scalable, open-source Web application called RUBiS. However, we believe our approach is generalizable to other types of applications deployable to the cloud. The paper is organized as follows: in Section 2 we define the concept of Profiling-as-a-Service. In Section 3 the technical details of our framework are described. Section 4 shows some evaluation of our implementation of the service, Section 5 goes over the related work, and finally we conclude in Section 6.

2 Profiling as a Service (PraaS)

Profiling in the cloud is important for closely metering resource usage of software and inferring the corresponding financial implications. Applying traditional models of offline profiling for monitoring and provisioning of resources is not effective for applications migrated to the cloud. This is mainly due to architectural differences before and after deployment to the cloud and the heterogeneity of various cloud infrastructures.

To illustrate the benefits of integrating the profiling process with the cloud, consider a typical 3-tier throughput-intensive auctioning Web application. We use the

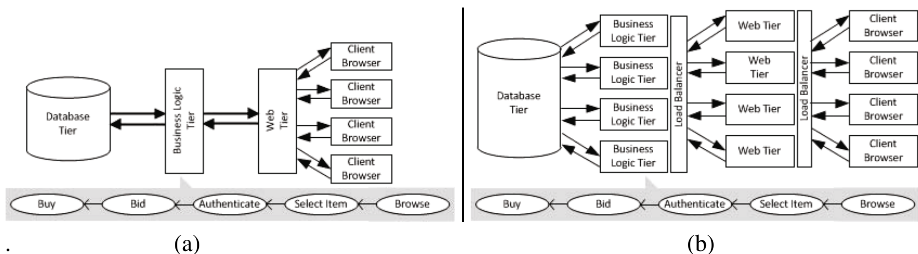
example of RUBiS [1,6], an open-source benchmark which simulates the activities of an e-commerce auction site. The original architecture of the system consists of a Web tier serving as the entry point for the application, a business logic tier containing the business logic of the application (e.g., searching, commenting, bidding, buying, authentication, and browsing of items as shown in Figure 1a-bottom), and finally a database tier to persist the transactions.

Figure 1b shows a potential architecture of the application after deployment to the cloud. As can be seen in the picture, several VM instances of the business logic tier and the Web server tier are instantiated and are placed behind load balancers. Clearly, profiling and monitoring of resources for the original application (Figure 1a) would not be helpful in understanding the behavior of the migrated application (Figure 1b). Consequently, resource usage footprints of the application in the cloud can be more effectively analyzed if profiling happens in the cloud.

When re-architectured for the cloud, all the instances in the Web server tier and the business logic tier are placed behind load balancers and hence their addition, removal, or modification stays transparent to the end-user clients of the application. These changes may only come to the attention of the end-user clients as response time delays or throughput alterations. Subsequently, as long as throughput shortfalls or response time delays are not significantly noticeable to the end-user clients, adaptive profiling strategies can be effectively blended into the overall behavior of the application.

2.1 Adaptive Resource Profiling in the Cloud

Adaptive profiling has been utilized in the past by many researchers [2,7,8,9]. Those efforts usually rely on duplicating the code blocks in an application, keeping an original version of the code along with an instrumented version. Upon occurrence of some triggering event, the instrumented and non-instrumented code are swapped [7], taking advantage of certain low-level code hot-swapping features available for some programming languages. In other efforts, instrumented and non-instrumented code are executed on different processors on one single machine [15]. In contrast, our high-level service performs adaptation at the granularity of operating system VM instances in the cloud. As such, our approach is compatible with a wider range of heterogeneous instrumentation strategies and programming platforms.



. (a) (b)

Fig. 1. A 3-tier Web application (a) before deployment to the cloud; and (b) after deployment to the cloud with potential architectural changes after cloud deployment

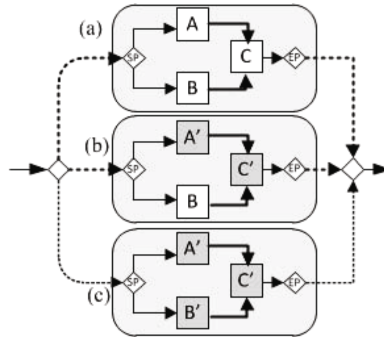


Fig. 2. Request flow through three operating system (OS) VM instances deployed from different VM images by the profiling service. The figure shows an illustrative example (i.e. the number of VM instances and application modules varies by application). This figure shows VM images with: (a) no instrumentation; (b) partial instrumentation; (c) full instrumentation. SP and EP define the start point and the end point for request flow in an application instance. A, B, and C represent different modules (e.g. classes) of the application. The shaded modules are the instrumented duplicates of the original ones for the application.

Figure 2 demonstrates the flow of a client request through an application using our profiling service. In the figure, several cloud-based VM instances of the example application are shown, processing client requests behind a load balancer. Each VM instance (a, b, or c) represents a previously imaged operating system with our custom-instrumented version of the application, deployed on a hypervisor in the cloud. For illustration we show a scenario with three different versions: (a) an instance with no instrumentation, (b) an instance with some instrumentation, and (c) an instance with full instrumentation. Our profiling service manages a repository of such VM image versions with their differing levels of profiling instrumentation. Each VM in the repository hosts a copy of the application instrumented statically right after the upload time of the original application. As described next, by utilizing declarative policies specified by developers and by monitoring certain QoS parameters, the service makes adjustments to the number of these different VM image types deployed for the application. This allows to tradeoff between performance and business requirements by adjusting the ratio of instrumented VM images to the non-instrumented ones.

2.2 Constraint-Guided Profiling Adaptation

Two major QoS requirements for Web applications deployed in the cloud are *performance*, particularly how it is perceived by the end-user clients (i.e., throughput and response time), and *cost of deployment*. Any effort to integrate profiling into the lifecycle of a deployed application to the cloud should actively respect these QoS requirements. However, supplying resources to boost the performance of an application deployed to the cloud will result in extra charges billed in monthly cycles to the clients of the cloud.

Given an upper limit for the target cost of deployment in one billing cycle in the cloud, C_t, a target performance requirement (e.g., specific throughput or response time)

for the application, P_t, and the expected performance P_{inst} after deploying a *fully-instrumented* application on a VM instance in the cloud, our adaptive swapping strategy will eventually ensure that the following inequalities hold:

$$m \times (P - P_{inst}) \geq P_t \quad (1)$$

$$C_{inst} < C_t \quad (2)$$

where m is the total number of VM instances leased from the cloud; P is the performance measure for the target application on a single VM in the absence of the instrumented code; and C_{inst} is the overall cost of running the application (in any of the non-instrumented, partially-instrumented, and fully-instrumented modes) in the cloud during a single billing cycle.

Our current strategy for implementing the above measures is based on a simple heuristic which increases the number of VM instances by a rate of $\lceil \alpha \times VM_{inst} \times (P_{inst}/P) \rceil$ where VM_{inst} is the number of instrumented virtual machines ($VM_{inst} + VM_{noinst} = m$), and α is a constant. In case the cost of instrumented deployment with increased number of VMs (C_{inst}) exceeds the previously set threshold C_t, we revert the instrumented instances back and incrementally replace the VMs running the instrumented code (VM_{inst}) with VMs running non-instrumented code (VM_{noinst}) until Inequality (2) holds again. Consequently, the current algorithm always prioritizes cost constraints to the expected performance measures. In other words, the algorithm can be thought of as a simple state machine. As long as the overall performance does not violate Inequality (1), the machine stays in an acceptable state. Once Inequality (1) is violated, the algorithm tries to bring the machine back to an acceptable state by adding more VMs or replacing instrumented VMs (VM_{inst}) with non-instrumented ones (VM_{noinst}). The state machine stabilizes under one of the following conditions: *i)* adding extra VMs brings the performance requirements back to normal without exceeding cost constraints of Inequality (2); *ii)* reverting some of the VM_{inst} machines to VM_{noinst} machines brings the performance requirements back to normal while Inequality (2) holds; or *iii)* All running machines are VM_{noinst} machines and while Inequality (1) is not satisfied, addition of another VM_{noinst} will violate Inequality (2).

In our current implementation, we translate application performance to the average application response time for requests. Hence, in Inequality (1), $P = RT$ where RT indicates the average response time when the application is in *no-instrumentation* mode; and $P_{inst} = RT_{inst}$, where RT_{inst} indicates the average response time degradation when the application is under *full instrumentation*.

For the cost of deployment, currently we consider C_{inst} equal to the total cost of application deployment (i.e., $\sum^m Cost(VM)$) during one billing cycle as defined by each public cloud provider (we provide details for Microsoft's Azure cloud in our evaluation). At this stage, we ignore other costs, e.g. the inbound and outbound communication costs and the costs of storing data in the cloud.

3 Technical Details

Now we turn to the specific details of our PraaS system starting with our policy specification support (Section 3.1), system architecture (3.2) and some implementation details for our specific prototype (Section 3.3).

3.1 Profiling Service Policy Specifications

To effectively expose profiling as a service to system architects, we wanted to provide a declarative policy model for controlling service parameters. Our current implementation allows for two sets of policy requirements to be specified: *Profiling Requirements* & *QoS Requirements*.

Profiling Requirements. System architects can define the level of granularity and the type of profiling that they want to be applied to the application during the execution of the application. The profiling requirements and specifications can be modified arbitrarily and even during the execution of the applications. For the level of granularity, they can choose instrumentation strategies to apply to the full application or a specific set of modules, classes, and methods in an application. They can also decide on the type of instrumentation, e.g., *CPU usage* or *Data Exchange* (described further in Section 3.3) and the scope of profiling. The scope of profiling can be defined as either *internal* or *external*.

Internal profiling only measures information internal to the elements of a module (e.g., its components, classes, and methods) while external profiling collects information from inter-module interactions in the application. Figure 3a shows a sample policy for RUBiS.

```
<profiling-spec>
  <!-- instrumentation constraints -->
  <instrumentation-map>
    <unit name="rubis.auth">
      <type>module</type>
      <profile>
        <mode>CPU</mode>
        <mode>Data</mode>
      </profile>
      <scope>internal</scope>
    </unit>

    <unit name="rubis.buy.BuyItem">
      <type>class</type>
      <profile>
        <mode>CPU</mode>
      </profile>
      <scope>internal</scope>
    </unit>

    <unit name="rubis.bid">
      <type>module</type>
      <profile>
        <mode>Data</mode>
      </profile>
      <scope>external</scope>
    </unit>
  </instrumentation-map>
```

(a)

```
  <!-- quality of service requirements
-->
  <qos-requirements>
    <cost>
      <vm-cost>2000</vm-cost>
    </cost>
    <performance>
      <resp-time>500ms</resp-time>
    </performance>
  </qos-requirements>
</profiling-spec>
```

(b)

Fig. 3. A sample instrumentation map defining (a) the modes of profiling for different modules in RUBiS and (b) QoS constraints

QoS Requirements. We also enable system architects to define their QoS constraints for instrumentation and profiling. QoS requirements are taken into consideration when ensuring Inequalities (1) and (2). As mentioned earlier, we consider a defined response time (RT_t) in milliseconds for the performance constraint of deployed Web applications and the upper limit dollar amount for leasing VMs from the cloud as the cost of deployment (C_t). Our framework supports extending these constraints with performance measures such as *throughput* or *database transactions*, and cost measures including *inbound/outbound communication costs*, and *data storage costs*. Figure 3b shows a sample QoS specification used in our RUBiS case-study.

The policy requirements of the developer are formulated into an *Instrumentation Map* document stored and loaded to a service *master node* as we describe next.

3.2 System Architecture

The architecture for our PraaS concept extends the architecture of a Web application deployed to the cloud, similar to the one in Figure 1b by adding a *master node* to each tier in the application that sits behind a load balancer. The master node encapsulates the core of the service and is loosely coupled to individual applications, communicating through an interface that specifies the exchange of profiling data and control messages. As the low-level instrumentation of code must be platform specific, this part of the service is isolated into a customized *profiler agent* colocated with each VM instance.

The profiler agent is in charge of collecting information about an application instance and reporting the collected results back to the service master node. The master node aggregates the results from all the agents and checks the validity of Inequalities (1) and (2) during the execution of the application.

As shown in Figure 4, the master node consists of the following five components: a *Profiler Specification Engine*, a *Policy Controller*, an *Instrumentation Map*, a *Reconfiguration Engine*, and a *Result Aggregator*. The profiler specification module allows the system architect to define the required profiling specification (as in Figure 3). Once the specification is loaded to the master node it is used by the master node to initiate the policy controller engine based on the `qos-requirements` part of the profiling specification, and to provide an instrumentation map. The instrumentation map is then communicated to each *Profiler Agent* to orchestrate the profiling behavior among all instances of the application.

The *Profiler Agent*, deployed together on each application node, has two components: *i)* a platform (e.g. Java, C#, etc..) specific component which takes care of instrumentation of application code, and *ii)* a *service integration module*. The SIM mediates communication of profiling data to the service master node. Through communication with the master node, SIM receives the instrumentation map specified by the system architect from the master node. The SIM then coordinates the loading of a VM image with the appropriate instrumentation.

During the profiling process, the performance on each application node gets reported to the SIM and the SIM periodically updates the master node about the status of the running application on its VM_{inst}. The master node aggregates the results from all application nodes and decides about potential reconfigurations for each node in the deployment. Upon a need for change in profiling, the SIM manages stopping and starting

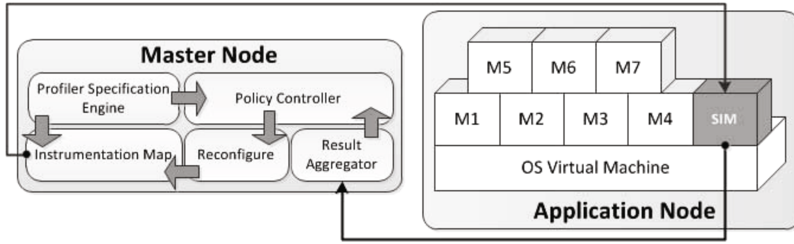


Fig. 4. Integration of Master Node and Application Node into the framework. M1 to M7 represent the classes/modules for the application and SIM is the service integration module.

new VM instances with the required instrumentation. Once the application is back to meeting all constraints, the SIM informs the master node and the master node coordinates the profiling process among all the running instances of the application again.

The master node can change the mode of profiling to one of the three already explained modes of *no instrumentation*, *partial instrumentation*, and *full instrumentation*. The adaptive switching of the profiling mode at this point is done by stopping the application on a target VM instance, replacing it with an application image in a different profiling mode, and starting the new instance. The process of mode switching is done for one instance at a time in order to minimize performance degradation caused by removing one instance of the running application. In addition to mode switching during profiling, the master node regulates the type of profiling to be performed for different instances of the application and also enables updates to the profiling process to be performed by system architects while the application is deployed in a production environment.

Our implementation for PraaS works independently of the type of instrumentation mechanism used by the profiler agents for deployed applications. Instrumentation strategies ranging from memory-leak [8] and performance bottleneck [4] detection to security related taint tracing [13] and resource usage monitoring [11,12] could be integrated into our framework. However, for our current implementation, we have particularly focused on monitoring resource utilization by different components of the application.

3.3 Prototype Profiling Support

While the service exposed by the master node is agnostic to specific platforms being profiled, a customized profiling agent is required for each programming platform (e.g. Java, C#, etc..) to be supported. Our current implementation supports Java profiling and we provide a profiler agent on top of The Java Interactive Profiler (JiP). JiP is a code profiling tool that supports performance timing, fine-grained CPU usage profiling to the level of classes and packages and requires no native code to enable profiling. JiP uses the ASM [5] library to provide manipulation, transformation, and analysis of Java classes at the level of byte code. We used the combination of JiP and ASM to collect information on CPU usage and data exchange between code blocks.

CPU Usage Profiling. CPU usage profiling is achieved simply by adding performance timers to the beginning and end of each function in the application. This is done by rewriting the Java bytecode for the function to include a `System.nanoTime()` timer.

Data Exchange Profiling. In order to make decisions on how to optimally partition software components across VM hosts in a cloud infrastructure, software developers can use profiling to determine the costs of information exchange between distributed components. Our instrumentation measures data exchange between software components by monitoring the size of remote function call arguments and return values. For local intra-VM method calls, such arguments and return values are typically passed by reference. So, in the case where developers are considering partitioning a local function into a remote function call, our framework will provide details on the size of the equivalent serialized data for each referenced argument or return value. This instrumentation strategy gives application developers a chance to measure data exchange in a monolithic application before deciding on the actual distribution.

4 Evaluation

We evaluated our current implementation against a case-study of the RUBiS benchmark.

As discussed earlier, RUBiS implements the basic functionality of an auctioning Web site following a 3-tier Web architecture with eleven components: a front-end Web server tier, nine business logic components (`User`, `UserTransactions`, `Region`, `Item`, `Category`, `Comment`, `Region`, `Bid`, and `Buy`), and a back-end database tier. Several implementations of RUBiS exist, but for our evaluation we used its Java Servlet implementation that makes use of the Hibernate middleware to provide data persistence. We deployed RUBiS together with our profiler agent modules on small instances of Microsoft's Windows Azure cloud platform. Each small Azure instance is equipped with a 1.6GHz CPU and 1.75GB of memory.

On each small Azure instance, we deployed the Web server along with all business logic components of RUBiS. For the database server, we used a 5GB SQLAzure database instance running SQLServer 2008 R2. When deploying the application on more than one instance, the Azure platform automatically places the instances behind a load balancer and distributes the load across all existing instances.

To provide a realistic client workload we used the RUBiS client simulator that comes bundled with the RUBiS benchmark [1]. The simulator was designed to operate in either a browsing mode or in a buy mode. In the browsing mode only browsing requests for items, users, comments, bids, etc. are launched. In the buy mode in addition to the browsing requests, requests to authenticate, bid on an item, or purchase of an item are also made. In our experiments, we used the clients in the browsing mode unless otherwise mentioned. Clients were launched from two machines, each equipped with a dual core 2.1GHz CPU and 4GB of memory.

4.1 Measuring Profiling Overhead

We modified the RUBiS client so that each client would generate requests at a pace of one every 125 milliseconds. To set a base line, in Table 1, we show the throughput and response time when only one single client launches requests to a single instance of RUBiS deployed on Windows Azure. We collected the data for both the profiling and the non-profiling modes. For the profiling mode, the entire set of components on the Azure instance were profiled to collect CPU usage information (cf. Section 3.3) and in the non-profiling mode, no profiling data was collected.

Table 1. Baseline throughput and response time when one client launches requests to a single instance of RUBiS deployed on one small Azure instance

	Throughput (req/sec)	Response Time (millisec)
Profiling Mode	2	432
Non-profiling Mode	6	74

From Table 1, we see the overhead of instrumentation in an isolated case. To mitigate this overhead we need to amortize it over our system. So next, in order to measure the effects of delegating profiling to a subset of all instances running an application, we made two deployments of RUBiS, one on 4 and another on 6 small Azure instances. We measured the change in response time and throughput when the number of instances running the profiling process goes from no instance (i.e., a profiling ratio of 0) to all instances (i.e., a profiling ratio of 1). A moderate load was generated on all instances in both deployments by launching 100 clients to perform 1000 transactions during a period of 5 minutes.

As Figure 5 shows, although throughput does decrease and response-time does increase as we increase the VM profiling ratio, our approach does mitigate the performance overhead of profiling. In particular, by keeping the ratio of instances that are profiled to less than 0.5 we are able to maintain throughput and response-time close to the non profile case, i.e. 200 requests per second. Only after we increase the ratio of profiled to non-profiled past 0.5 does performance significantly decline. In essence, this indicates that the Azure load balancer does a good job of intelligently redirecting requests to less busy nodes of the deployment.

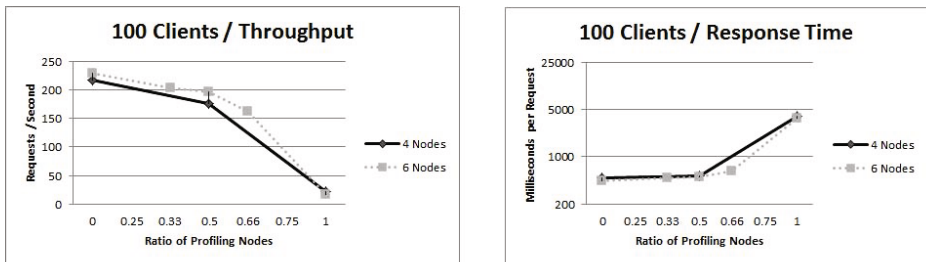


Fig. 5. Throughput and Response Time when 4 and 6 clients with different modes of profiling are placed under moderate load of client requests

4.2 Measuring Profiling Accuracy

To measure the accuracy of the PraaS system, profiling data collected at different profiling ratios were compared against the perfect profiling data (i.e., where all nodes were doing profiling), and we computed the corresponding accuracy metrics. We used the *Overlap Percentage* metric suggested by Arnold and Ryder [2] to measure the accuracy of collected profiling information. As described in [2], the overlap of two profiles represents the percentage of profiled information weighted by execution frequency that exists in both profiles. Obviously, the overlap percentage metric is a function of the diversity of requests for which profiling data is collected. In order to assess how the diversity of requests affects the overlap percentage, we collected the overlap percentage data in two modes: *i) Single Request Mode*, where RUBiS clients only launched *BrowseCategories* requests, and *ii) Multi-Request Mode*, where RUBiS clients launched all sorts of browsing requests, from browsing item categories, to browsing and searching items, browsing user information and their bid and buy histories. Table 2 shows the result of measuring overlap percentage for each of these deployments.

Table 2. The Overlap Percentage measure for accuracy of profiling information subject to the diversity of requests. Table 2 provides the overlap percentage measures for all profiling ratios of Section 4.2 for which we have throughput and response time collected.

	Single Request			Multi-Request		
Profiling Ratio	Num Samples	Num Unique Methods	Overlap %	Num Samples	Num Unique Methods	Overlap %
2 of 4 (0.5)	6.07×10^7	6338	99.65	2.74×10^7	7094	97.62
2 of 6 (0.33)	3.00×10^7	6325	99.03	3.07×10^7	6992	91.53
3 of 6 (0.5)	5.78×10^7	6329	99.10	4.35×10^7	7014	92.21
4 of 6 (0.66)	9.19×10^7	6339	99.34	5.32×10^7	7092	93.13

As expected, increasing the number of instances increases the number of samples taken during profiling. However, as we discussed earlier, increased diversity in types of requests results in a lower overlap percentage between partial profiling and full profiling. Since RUBiS is only a small representative of potential enterprise Web deployments, we expect deployments of larger applications to result in lower overlap percentages when doing partial profiling. Nonetheless, an increase in the number of nodes clearly brings the collected profiling results closer to a full profiling deployment.

4.3 Stress Testing of the Deployment and Financial Implications

In order to stress test the application, we ran three deployments of RUBiS using 4, 6, and 8 small instances on Windows Azure. Each deployment was tested with batches of 1600 and 3200 clients launching 1000 requests to it during a period of 5 minutes. Figure 6 shows the throughput and response time when 1600 and 3200 clients send requests to the deployed RUBiS application. We summarize the implications of these results next.

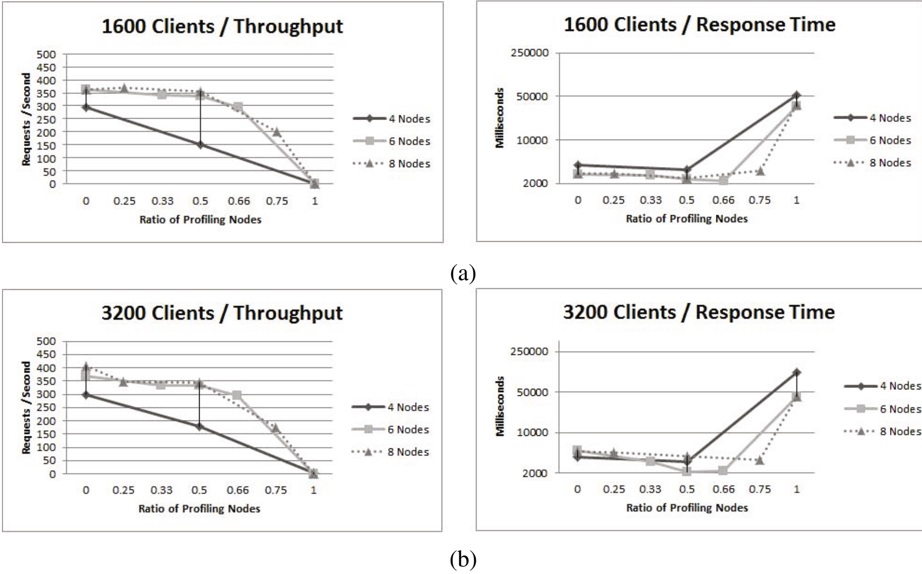


Fig. 6. Response time and throughput for (a) 1600 clients, and (b) 3200 clients; sending requests to RUBiS deployed on 4, 6, and 8 small Azure instances

In order to assess financial implications, in Table 3 we calculate the hourly and monthly costs of deployment for each of the three deployments above.

Table 3. Deployment costs for RUBiS on various number of small Azure instances

	Deployment Costs (USD)		
Num Instances	Hourly	Monthly	Yearly
4	$0.48	$345.6	$4147.2
6	$0.72	$518.4	$6220.8
8	$0.96	$691.2	$8294.4

4.4 Evaluation Summary

To summarize these findings we see that our approach of "Profiling-as-a-Service" does indeed provide a way to more accurately profile an application while respecting performance and cost constraints. In particular, it should be noted that the throughput could be kept above *300 req/sec* with 8 nodes of 0.5-profiling-ratio and marginal throughput loss (i.e. within 1% of a baseline with no profiled instances), or 6 nodes of at most 0.66-profiling-ratio (i.e. within 20% of a baseline). With a 0.75-profiling-ratio for 8 nodes, we achieved a lower throughput compared to the 0.66-profiling-ratio with 6 nodes, however the response time measured with 8 nodes was significantly smaller than the response time measured with 6 nodes. Conversely, from a financial perspective running the deployment on 6 nodes costs almost $173/month less compared to running 8 nodes. Again this demonstrates the flexibility our approach offers, allowing developers

to trade-off throughput against response time, and then factor in cost. During our experiments with 3200 clients sending requests to our RUBiS deployment, our CPU resources reached their limits and hence, even though we expected to see a higher throughput, the throughput stayed the same as for our experiment with 1600 RUBiS clients. It is worth mentioning that in our current implementation, transitions from 4 nodes to 6 nodes to 8 nodes were done manually. It is part of our future work to make this transition dynamic and based on the requirements of the target application.

5 Related Work

Profiling of service-oriented applications forms the basis of much existing work in both on-line monitoring of SLAs [3] and also the autonomic management of services. While the scope of previous work is too large to cover here in depth, we can say that compared to previous work, this paper focuses on the low-level support of profiling in the cloud environment. Previous work, on the other hand, has focused on more specific strategies for utilizing runtime profiles i.e. how to analyze and react to such profiles. Thus previous work did not cover the cloud-based adaptive instrumentation provided by our profiling service. This research simply supports an efficient profiling mechanism at the systems level. We did not address any specific policies for utilizing profile data, as we sought to provide profiling as a generic reusable service.

One of the first approaches to directly address the performance problem of on-line profiling was presented by Arnold and Ryder [2]. They use a compiler-based approach which duplicates the code for each method into instrumented and non-instrumented versions. Additionally, the compiler inserts certain "switches" in the code to allow execution to be dynamically re-directed along either instrumented or non-instrumented paths. In similar efforts Dmitriev [7] and BEA Systems' JRockit [14] use modified compilers which enable dynamic code hot-swapping to reduce profiling overhead. The approach in this paper also uses code duplication to manage profile overhead. However, our research work is different in that duplication occurs at the level of entire VM-instances. This allows a more general technique, independent of the details for specific compilers. We take advantage of the transparency afforded by cloud platforms to shield end-users from the details of swapping VM-instances dynamically. In this sense, our work is similar to the work done by Wallace and Hazlewood for SuperPin [15]. In SuperPin the authors slice the non-overlapping pieces of code into separate execution threads and run them in parallel and on multiple processor cores, gaining significant performance improvements through the added parallelism. The main difference as mentioned earlier is that we benefit from parallelism at the level of Operating System VM instances by spreading the instrumented code for the application across multiple machines in the cloud rather than using different cores on a single machine. Benefiting from the elasticity of cloud and by increasing the number of machines used for profiling, we can overcome the limitations to the degree of parallelism caused by the limited number of cores when running the instrumented software on one single machine.

Adaptive bursty tracing (ABT) [8,9] is a particular technique built for the collection of traces in profiled applications. Since trace logs can grow to enormous sizes, most profiling approaches use sampling to limit log sizes. The problem with sampling is that

it may capture very limited information about infrequently executed code. However, as authors claim, often the worst bugs and performance bottlenecks hide themselves in such code. ABT ensures that detailed traces are generated for infrequently running code, by providing a sampling rate inversely proportional to code execution frequency. In the future we will explore applying ABT to our the context of our distributed service.

AjaxScope [10] implements a JavaScript instrumentation proxy to provide monitoring and profiling of code that executes in an end-user's Web browser. This allows on-line profiling in a distributed context, where code is deployed on a server, yet later executed on the client. Traces of client behavior are periodically uploaded to the server infrastructure for analysis. Similar to our research, AjaxScope targets a distributed computing context. However, where AjaxScope focuses on client behavior, this research is focused on the server-side. This distinction changes the kind of techniques which are applicable for providing transparency. In AjaxScope, transparency is provided to clients through an instrumentation proxy whereas our research leverages the flexibility of OS VMs used in a cloud computing context.

6 Conclusion

This paper described the design and implementation for a cloud-based profiling service. This service was motivated by the need to manage tradeoffs between three important factors in the deployment of cloud services and applications: performance, cost, and accuracy of monitored profile data. We showed the validity of the approach in the context of an existing Web application deployed to the cloud. The results showed that while the reduction of profiled instances through adaptation did reduce the accuracy of profiling, it also improved performance and reduced cost. More importantly, accuracy degraded at a much slower rate than performance and cost improved.

Acknowledgements. We would like to thank Microsoft Windows Azure team, and particularly Ori Amiga, for providing us with access to resources on Windows Azure.

References

1. Amza, C., Chanda, A., Cox, A., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W., Cecchet, E., Marguerite, J.: Specification and implementation of dynamic Web site benchmarks. In: IEEE International Workshop on Workload Characterization, pp. 3–13 (November 2002)
2. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 168–179. ACM, New York (2001)
3. Baresi, L., Guinea, S., Pasquale, L.: Integrated and Composable Supervision of BPEL Processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 614–619. Springer, Heidelberg (2008)
4. Brear, D., Weise, T., Wiffen, T., Yeung, K., Bennett, S., Kelly, P.: Search strategies for java bottleneck location by dynamic instrumentation. IEE Proceedings - Software 150(4), 235–241 (2003)
5. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems, Grenoble, France (November 2002)

6. Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., Zwaenepoel, W.: Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 242–261. Springer, Heidelberg (2003)
7. Dmitriev, M.: Profiling Java applications using code hotswapping and dynamic call graph revelation. ACM Sigsoft Softwre Engineering Notes 29(1), 139–150 (2004)
8. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XI, pp. 156–164. ACM, New York (2004)
9. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling. In: The 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO4), pp. 117–126 (2001)
10. Kiciman, E., Livshits, B.: AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 17–30. ACM, New York (2007)
11. Luk, C.K., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of Programming Language Design and Implementation Conference, pp. 190–200. ACM (2005)
12. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer 28(11), 37–46 (1995)
13. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed System Security Symposium, NDSS (2005)
14. Systems, I.B.: Jrockit (August 2008), `http://www.bea.com/jrockit/`
15. Wallace, S., Hazelwood, K.: Superpin: Parallelizing dynamic instrumentation for real-time performance. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 209–220. IEEE Computer Society Press, Washington, DC, USA (2007)