2014 IEEE International Conference on Services Computing

Performance Antipatterns: Detection and Evaluation of their Effects in the Cloud

Vibhu Saujanya Sharma
Accenture Technology Labs
Accenture, Bangalore, India
Email: vibhu.sharma@accenture.com

Samit Anwer
Indraprastha Institute of Information Technology, Delhi
IIIT-Delhi, New Delhi, India
Email: samit1274@iiitd.ac.in

***Abstract*—The way an application is designed and certain patterns thereof, play a significant role and might have a positive or a negative effect on the performance of the application. Some design patterns that have a negative effect on performance, also called performance antipatterns, may become important when evaluating migrating the application to the Cloud. Although there has been work done in the past related to defining performance antipatterns, there has been none that highlights the importance and effects of these performance antipatterns when an application is migrated to Cloud. In this work we present an approach to automatically detect important performance antipatterns in an application, by leveraging static code analysis and information about prospective deployment of the application components on the Cloud. We also experimentally show that these antipatterns may become prominent and pull down the application's performance if the application is migrated to the Cloud. Our results show that the performance of the parts of the application with such antipatterns suffer significantly and hence, the detection of these antipatterns has an overarching significance in the domain of software development for the Cloud. The approach we present here has also been implemented in a prototype cloud migration assessment tool.**

***Keywords*—Cloud computing, Performance antipatterns, PaaS, Cloud migration, CloudFoundry**

I. Introduction

Software performance is a critical aspect of any software system and it is regarded as a valuable system characteristic. With the adoption of Cloud computing, there is a buzz of expectations that enterprises have from cloud. Among them, a common one is that, migrating existing applications to Cloud will result in high performance and scalability. However, this is also an area where a new set of capabilities and challenges have emerged in the recent times. While Cloud computing promises elastic capacity and scaling, applications being migrated to Cloud should also be able to exploit this elasticity and be inherently scalable. The performance of a software system is dependent on various aspects like the software architecture and design, the hardware infrastructure on which it is deployed, the manner of deployment and the workloads that the application will endure. So, to assume that simple lift and shift of one's application to Cloud would result in high software performance is trivializing a much more complex problem.

The inherent architecture and design of the application is perhaps the most important aspect affecting overall performance. The design of a software system governs how it utilizes its environment and the underlying infrastructure. Thus while assessing migration of an application to a Cloud platform, it is essential to look at the way different modules of the software system interact with each other and how they are implemented internally. Certain well known ways of designing different types of software systems are typically called *Design Patterns* [1]. Similarly, *Antipatterns* are typically a commonly used set of design and coding constructs which might appear intuitive initially, but eventually may be detrimental to one or more aspects of the system. Many antipatterns with respect to different aspects of the system (like performance, maintainability, etc.) are well documented and in many cases, ways to re-factor and mitigate them are also present. Specifically, the work presented in [2], [3], [4] outlines various antipatterns with respect to performance of software systems, and thus is important for the current discourse.

Performance antipatterns are patterns of application design and implementation that degrade the systems performance. These may manifest in a systems control flow, deployment, code/implementation, data flow, memory allocation, thread allocation, scheduling, etc. Since migrating to Cloud is bound to affect one or more aspects of the system in these dimensions, performance antipatterns become important in the Cloud context. Specifically, certain deployments - i.e. distribution of system modules off to different domains/machines, may cause certain antipatterns to become prominent and manifest themselves as performance bottlenecks of the system. For example, a nested set of consecutive calls to different modules (also called *Circuitous Treasure Hunt* [2]), maybe benign from a performance point of view if all those modules are locally situated, but this becomes a potential bottleneck if those modules are distributed. Hence, we advocate refactoring and tuning the code before migration vis-a-vis a blind migration. Note that detecting such antipatterns is typically a manual effort intensive task. Moreover, missing these before actual migration to the Cloud may result in expensive rework much later, when they manifest into poor performance of the migrated system.

In this paper, we look at the challenge of assessing migration to Cloud platforms from the point of view of performance. Our approach is based on detecting performance antipattern signatures in application code and then examining them from the perspective of becoming significant if the application is deployed on Cloud. This work presents our approach for detecting three types of performance antipatterns, namely; Empty Semi Trucks, Circuitous Treasure Hunt and Blob. Through experiments we show that parts of the software system where we detect these antipatterns perform poorly on Cloud. To the best of our knowledge our approach is unique in that no

978-1-4799-5066-9/14 $31.00 © 2014 IEEE
DOI 10.1109/SCC.2014.103

other work has attempted to automatically detect performance antipatterns in the context of assessing migration to Cloud. The experimentation of evaluating the effects of performance antipatterns on the scaling of application's execution time while migrating to Cloud is also unique and novel.

The paper is organized as follows: In the next section, we present an overview of performance antipatterns and why they become important in the context of Cloud Computing. We present our approach for detecting various performance antipatterns in Section 3. In section 4, we describe the prototype implementation called PAD, which has been added to our prototype migration assessment tool called MAT [5]. We then describe our experimental setup and some early experimental results showing the effects of performance antipatterns in Section 5. Finally, we present our observations and conclude the paper in Section 6.

II. Importance of Performance Antipatterns in context of the Cloud

Capabilities like on-demand horizontal scaling and load balancing allow cloud platforms to promise elasticity and high performance. However, this promise can easily be misinterpreted as "Putting my application on Cloud is guaranteed make it perform much better! " [6]. This is obviously not true - there are a number of reasons for this. Firstly, depending upon the type of Cloud platform, the environment an application executes in on the Cloud will be different and at times it can be more 'limited' than the application's in-house environment. For example, it is very common for PaaS containers (DEAs[7], Dynos[8], etc), to be constrained in terms of the amount of memory and hard disk space. The processing capabilities are also usually not specified and are internal to the PaaS platforms. These too may be limited compared to dedicated servers on which the application would have been deployed on in-house/locally. A related issue is that of configurability - the ability to configure the execution environment may be very limited, specially on PaaS platforms, and for that reason one may not be able to leverage techniques for high performance (like clustering instances together otherwise done by say using Tomcat clustering [9]), which require tighter transactional bounds and semantics.

Finally, and importantly, the internals of the application and the way it is deployed on any Cloud platform heavily influences its performance. Deploying a poorly written application on Cloud will more often than not, result in sub-par performance. In other words, the application which has inherent limitations with respect to performance due to its internal structure and code will not scale the way it is expected to. We postulate that one of the chief reason for this is certain patterns of coding and design that have been known to be detrimental to software performance (called Performance Antipatterns [2]). The presence of these in certain parts of the application can prevent the application from fully utilizing the cloud environment. In certain other cases, the presence of antipatterns can cause the application to demand significantly larger amount of resources (memory, bandwidth, etc.) than it is planned or projected.

Thus, while migrating to a Cloud platform, one needs to carefully analyze the application for any performance antipatterns. Besides making the application sluggish and more resource hungry, in some cases, such patterns may prevent one from successfully utilizing the elasticity of Cloud. Antipatterns like Empty Semi-Trucks [2] and Circuitous Treasure Hunt become immediately critical in context of Cloud computing as they are based on inefficient or highly nested and frequent use of the network which may have been avoided. If left as such in the application, these will not only cause huge delays in terms of holding network connections inefficiently, but also incur significant costs. Others like Blob [2] will cause processing capabilities to be over utilized and thus would again require more powerful Cloud tiers, adding up to the overall costs. In some cases, Blobs in an application may also render a Cloud container (as in PaaS platforms) inadequate for deploying the application modules, thus preventing one from even migrating to Cloud.

Performance antipatterns were first introduced in [2] and then more antipatterns were introduced in [3], [4]. These papers presented the essential nature of each antipattern in terms of the description, the typical effects on performance and example manifestations of the same. These papers have been referred in subsequent works while referring to performance antipatterns; there however have been few automated approaches to finding performance antipatterns (specially from code). Similarly, there is no research work in the area of assessing the effects of such antipatterns in the Cloud computing context. Addressing these challenges forms the crux of this paper.

There have been some related work (though limited) in detection of design patterns automatically or semi-automatically from code. In [10], the authors detect patterns without using a pattern library. The precise implementation of the patterns need not be known in advance. They use a technique called Formal Concept Analysis, to find structural patterns in two subsystems of a printer controller. Their case study shows that it is possible to detect frequently used structural design constructs without upfront knowledge. In [11], the authors use an approach of detecting design patterns in the source code by using *code smells* and a set of pattern specifications. In [12], the authors give their approach to detect similarities in code. They present a highly configurable clustering workbench that allows the user to collect various source code features and then to select the code features used for clustering.They also present a comparison of outputs based on different combinations of code-features and algorithms.

The work [13] uses both static and dynamic analysis to identify various design patterns in the code. The authors use a tool called Recoder to read the source code and construct the abstract syntax tree (AST). They then perform static semantic analysis which resolves different names and types. The approach computes relations like define-use-relations, and provides all these entities and relations as an API. These program parts are then instrumented using the code manipulation interface of Recoder, which simply adds appropriate event generators tracing the dynamic execution of the candidates and providing runtime information. Recently, in [14], the authors convert the Software Architectural Model (UML annotated with MARTE, PCM) of application to a Performance Model like Queuing Network and identify performance antipatterns (from design specifications) in them using First order Predicate Logic to represent antipatterns and the defined thresholds for various features like maximum connections, messages, hardware utilization, etc. In [15], the authors propose a framework for

detecting performance antipatterns in component based systems using dynamic analysis, by applying techniques borrowed from the field of Knowledge Discovery in Databases (KDD).

The approach we follow in this work is to assess an application for presence of possible performance antipatterns to be helpful *before* an actual migration and deployment on a Cloud. Unlike some other works in this area, our focus is only on performance antipatterns and the assessment approach based on static analysis of application code coupled with a possible what-if analysis of the expected deployment of various parts the application. Note that we cannot assume to be able to perform a dynamic analysis on applications as they are yet to be migrated to a Cloud platform. The approach leverages parts of the work in [12] to abstract out code into a set of interacting clusters of classes. Subsequently we utilize user inputs on how these interacting clusters would be distributed across various nodes in the Cloud, and employ a threshold based analysis on detecting patterns of interaction in parts of the system that could potentially exhibit performance antipatterns. We presented a brief snapshot of some parts of our approach in [6]. However this paper presents a significantly more detailed and complete description of our detection approach as well as our experimental methodology and results, along with a wider overview of related work in this area. Moreover here we also present the details of the recently completed prototype which implements our approach.

III. Detecting Performance Antipatterns

Our approach for detecting antipatterns leverages static analysis to unearth patterns of interactions among various parts of the system along with different object-oriented attributes associated with the individual classes. Coupled with the information about how these parts will be distributed across the Cloud, we use thresholds to detect the potential performance antipatterns that may become prominent. At a high level, we can represent the approach as in Figure 1.

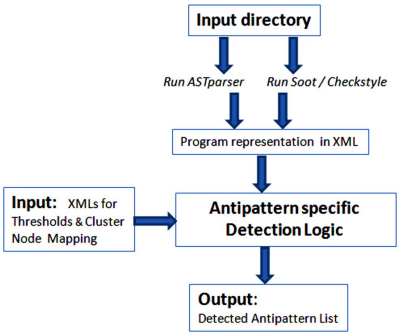

Fig. 1. High Level Diagram showing approach for Detection of Performance Antipatterns

The inputs are the application project directory ('Input directory'), the information about how different parts of the system will be deployed on distinct nodes ('Cluster Node Mapping'), as well as a set of configurable thresholds relevant for a particular class of antipattern. Note that we utilize a prototype implementation (CCW- Code Clustering Workbench) of the approach presented in [12] and further include a user input on how individual elements of this view will be distributed across Cloud nodes. The CCW tool creates clusters of classes based on Textual, Code and Structural features. It is just a way to represent the application in a modularized way. This representation is then a basis of annotating with the possible allocation of different clusters or modules of the system to different Cloud nodes - this is represented by 'Cluster Node Mapping'. We use this to find out, which Cloud node, a class is deployed upon. i.e. we can create a node to class map. This information is represented and consumed in an XML format. In this section, we discuss the details of the analysis approach for three performance antipatterns - Empty Semi-Trucks, Circuitous Treasure Hunt, and Blob.

A. Empty Semi Trucks

Empty Semi Trucks antipattern appears as a result of inefficient use of available bandwidth or an inefficient interface. It manifests itself in scenarios where a large number of requests are required to perform a task. For instance, in message-based systems, even if very small amounts of information is sent across in each message, the amount of processing overhead remains almost the same for each message regardless of the size of the message and thus, the processing overhead accrues as compared to a scenario where such information was communicated in lesser number of messages [4].

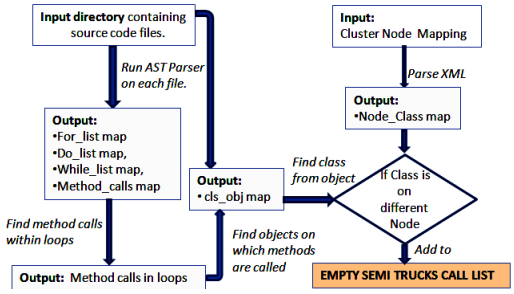

Fig. 2. Empty Semi Trucks Antipattern detection

As shown in Figure 2, to detect Empty Semi Trucks we use the ASTParser class of JDT API which allows us to find out all the method calls that are present within loops. We currently do not base our decision on the number of iterations in the loop because that is not determinable by a static analysis of code. We believe that such coding practices are a stereotype of Empty Semi Trucks. Note that such repeated calls may be necessary to the functioning of the system and may not be avoidable. However, the aim here is to unearth such patterns and let the decision of what to do with those rest with the experts.

We utilize the Cluster node mapping input to establish which set of interacting methods will lie on different nodes in a Cloud deployment. This is important since such a communication incurs network overheads as well as introduces latency in the execution, and thus this step prunes off the initial candidate set. If such calls are local in nature, they may not be detrimental to performance. So, the input directory is parsed to check whether the called method belongs to a class on a different node. If yes then the caller class is potentially an Empty Semi Trucks antipattern initiator.

B. Circuitous Treasure Hunt

As in the game of *Treasure Hunt* where one needs to move from one clue to another in search of hidden treasure, Circuitous Treasure Hunt (CTH) antipattern manifests itself in Object Oriented Systems where one object invokes a method in another object, then that object invokes an operation in another object, and so on until the ultimate result is computed and returned back one by one [2]. This typically entails huge overheads in creating and destroying intermediate objects as well as communication across the network. To detect CTH, one needs to find chain of such methods such that the number of nodes hopped across i.e. hop count, in the chain of calls exceeds a certain threshold. If the hop count exceeds the threshold then this chain of method invocation is deemed as Circuitous Treasure Hunt antipattern. Note that this threshold is highly context dependent. It may be very low (say just 3) for applications that are not supposed to be distributed in nature, or where such hops are extremely expensive (cost or time-wise). On the other hand, inherently distributed and complex applications spread across many domains may choose to keep this high.

Figure 3 shows the flow diagram for detecting Circuitous Treasure Hunt. As earlier, using the output of Cluster Node mapping information we can find out which node a class belongs to in the eventual deployment. This node class map can be used to count the number of hops in the chain of method calls as well. We use Soot framework to create a Jimple representation of the code [16]. Jimple represents the dependency of a class on other classes by their fully qualified names, thereby making it easier to resolve dependencies. By parsing the XML file generated by Soot, we create an adjacency matrix which is then used to find 'All Pair Possible Paths'. This is an exhaustive list of all possible paths of execution through the system. We then calculate the hop count for each such chain of method calls. Those chains, whose hop count exceeds the threshold are classified as Circuitous Treasure Hunt calls.

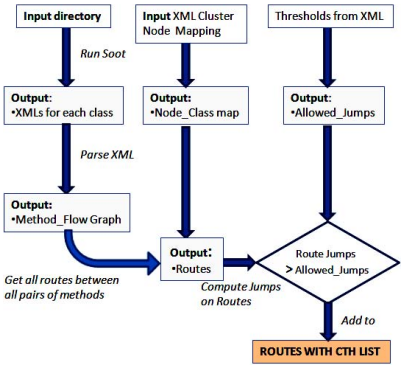

Fig. 3. Circuitous Treasure Hunt Antipattern detection

C. Blobs

A Blob as explained in [2] is the module "that performs most of the work of the system, relegating other classes to minor, supporting roles". Though these can be modules or packages, for simplicity, we assume these typically are complex controller classes that are surrounded by simple classes that serve only as data containers. Data Container classes typically contain only accessor operations (operations to get() and set() the data) and perform very little computation of their own. Further, Blob classes obtain the information they need using the get() operations belonging to these data container classes, perform some computation, and then update the data using the set() operations [2].

To detect the Blob antipattern we use four metrics in conjunction. The first one is *Class Fan Out Complexity* (CFOC) which gives the number of external classes a class is dependent on, second is *Number of Methods* (NOM) in a class, third is *Weighted Method Count* (WMC) which is the sum of cyclomatic complexities of methods in the class. This helps in highlighting the dominant classes that do much of the work. Finally, *Lack of Cohesion in Methods* using Henderson Seller method (LCOMHS) signifies whether or not the class has multiple responsibilities. High values of all these metrics tend to highlight a huge and bulky controller class. Typical thresholds for each of these can be found from existing literature - these are typically, 20 for CFOC, 7 for NOM, 20 for WMC and 1 for LCOMHS.

Another metric that we consider to detect Blob is Access to Foreign Data (ATFD). ATFD indicates whether a class, present on one node, accesses attributes from other classes present on a different node using accessor methods. To find classes that access foreign data, we target to find method calls that return values and then find the classes that such methods belong to (Data Container classes). We classify a class as a Data Container for the Caller class if:

1) It has complexity equal to the number of methods it possesses i.e. It has getter and setter methods only.
2) It is on a different node than the caller class.

For calculating the NOM, WMC, and CFOC metrics, we used the Checkstyle and JDT API. Checkstyle is a development tool that checks Java code for adherence to certain coding standard. [17] In Checkstyles configuration file, we can mention the threshold for Maximum Number of Methods, CFOC and WMC threshold for a class. Checkstyle outputs the violator classes of these thresholds. The ASTParser class of JDT API can be used to parse the source code and find out the line numbers where classes, variables, code constructs and methods are defined and used. By a single parse of the source code, we compute the LCOMHS as described in [18]. Note that we only include methods if they access at least one field and only include fields if they are accessed by at least one method in the class. The range of LCOMHS is from $[0, 2]$. Values of LCOMHS < 1 indicate that cohesion is quite good [19].

Figure 4 depicts a flow diagram for detecting a Blob class. First, the input directory containing the source code is parsed. This results in populating a number of data structures, which tell us about the various features of the code like class names, method names, field names, simple names, etc. and the corresponding line numbers where they are referred. From this information we can compute LCOMHS for every class. Secondly, on the input directory we run Checkstyle to find classes that violate thresholds of Number of Methods, 'Weighted Method Count', and 'Class Fan Out Complexity'. The classes that are common violators of these three lists and the LCOMHS list are deemed as Blob classes. Further, by one extra AST Parse over the Blob class, we then find the Data Container classes corresponding to the Blob class.

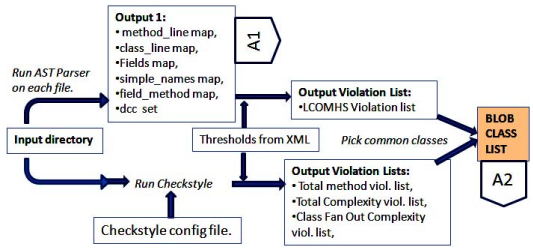

Fig. 4. Blob class detection

Figure 5 shows a flow diagram for detecting Data Container classes for a Blob class. If the two conditions for a Data Container class are met, then we can classify the called class as a Data container class corresponding to the Blob class.

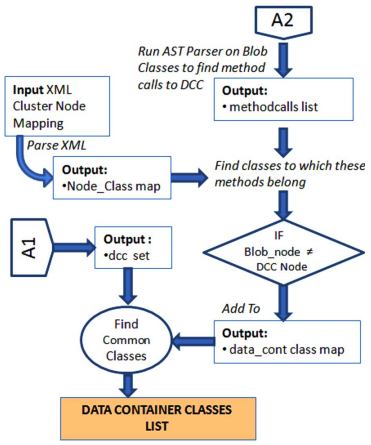

Fig. 5. Data Container class detection

IV. A Tool for Detecting Performance Antipatterns

We created a prototype implementation for detecting a few performance antipatterns using the analysis approach described in the last section to explore the feasibility of the same as well as gauge at results of the proposed analysis. The prototype is called PAD which stands for Performance Antipattern Detector and has been implemented in Java. This is an extension to our Migration Analysis Toolkit (MAT) which was presented in [5]

PAD uses MAT's initial step to first ask for the project details as shown in Figure 6. This is followed by entering the source code of the project to be analysed. The CCW tool is then run to create a preliminary set of clusters of various parts of the system. We present these clusters to the user and allow changes to these - the aim is to create units which with good cohesion and which are amenable to being deploy relatively independently of each other.

On the next screen (Figure 7), the user is asked to separate out the clusters based on where they would be deployed - Inhouse or on Cloud. Further for a Cloud deloyment, the user can further indicate if they would be on same or different nodes. As we discussed, this input helps to create the class node map, which is required to detect EST and CTH antipatterns.

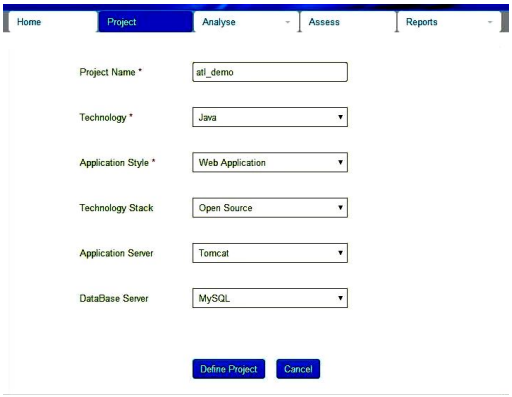

Fig. 6. PAD- Project Specification

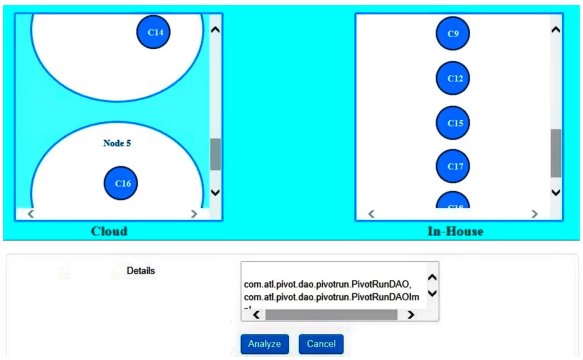

Fig. 7. PAD- Cluster Deployment Input

Figure 8 shows and snippet from the output of the tool. The antipattern shown here is CTH and the representation shows a cluster on a Cloud node communicating with a cluster which is in-house, which in turn depends again on a Cloud node. Such communications are the signature of CTH antipattern and ideally such disparate communicating clusters should be co-located.

V. Evaluation of Effects of Performance Antipatterns in the Cloud

Next we set out to evaluate the adverse effects performance antipatterns on the performance of the system, if it were to be migrated to a Cloud platform. We characterize performance by the execution time of different parts of the system. For the purpose of experimentation, we considered two antipatterns: Circuitous Treasure Hunt (CTH) and Empty Semi Trucks (EST), and emulated and inserted them in an antipattern free application. We also re-architected relevant parts of the application to be able to interact in a distributed scenario (i.e. when deployed across nodes locally (in-house) or on Cloud). For making our experiments more generic, we created two versions of these distributed applications - one which utilizes Rabbit MQ and other that uses web service calls for communication. We call these versions as the RMQ and WS versions in our descriptions. We then compared the scaling of execution time between control methods (i.e. methods without antipatterns) and the methods that manifest antipatterns to validate our hypothesis. The Cloud platform of choice used for the experiments was Cloud Foundry

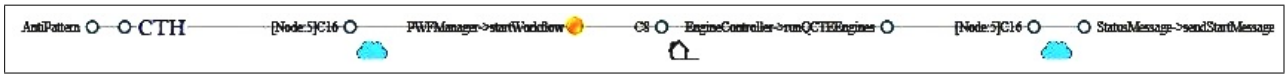

Fig. 8. PAD- A sample output showing a detected CTH antipattern

[7] as it is a popular open source PaaS platform and promises to be developer friendly. We present details of our approach here.

Experiment Design

For the purpose of our experiments we used the 'spring-petclinic' application, available on GitHub [20]. For Circuitous Treasure Hunt (CTH) antipattern, we customized the application by inserting CTH antipattern across parts of the original code. Specifically we made the application a 3-tier system with 3 sub-applications which can be deployed independently. The functionality is now split across the three sub-applications - the first is called the 'Client application', the second is called the 'Intermediate server application' and the third is called 'Server application' (the nomenclature is based on typical client server paradigm). For Empty Semi Trucks (EST), 'spring-petclinic' application was customized by inserting EST antipattern among the code. Then we compared the scaling of execution time of antipattern containing methods with the control methods. The 'spring-petclinic' application was split across two nodes in this case named as the Client sub-application and the Server sub-application. The data access parts for retrieving 'Vet' information in the original 'spring-petclinic' application was moved to the Server application. A controller class in the 'Client' makes the initial request to the Server application. The Server application responds back sending back one record at a time, thereby, not utilizing the bandwidth efficiently. This simulates the Empty Semi Trucks antipattern.

A. Circuitous Treasure Hunt

1) Test Bed:

- *Local Deployment:* The test bed for local deployment consists of three machines, hosting three different applications for simulating the Circuitous Treasure Hunt. We used Tomcat version 6 to deploy applications, since, Cloud Foundry also uses the same version. The first machine holds the Client application viz. 'spring-petclinic' merged with code for requesting the i^{th} record from the 'Vet' schema. This application initiates a request. Second machine holds the Intermediate Server (which also runs Rabbit MQ Server in cases when we use Rabbit MQ). The Intermediate Server forwards the requests to the third machine, which has the Server sub application that finds the i^{th} record by using 'id' field in the table 'Vet' and returns the response to the Intermediate Server which in turn returns the response to the Client, thereby, simulating Circuitous Treasure Hunt. One also need to make sure that you have added the 'mysql-connector' jar in Tomcats '/lib' directory and that Windows Firewall is turned off. The execution times are computed by injecting *'System.currentTimeInMillis()'* statements in the code and then subtracting Start time from End time. The in-house systems that we use for Local deployment in our experiments were Pentium Dual Core, 3.2 GHz machines with 2GB RAM and Windows 7 OS. Also, when using Rabbit MQ for RPC, we use default 'Exchange' and no queue bindings since we are using only single queue between a Consumer and Producer.
- *On Cloud:* The test bed on the cloud consisted of three execution environments, hosting the three different applications. The Cloud Foundry plug in for Spring Tools Suite was used to deploy applications. The user is prompted for a name for the application, URL and memory space that will be allocated to the application and finally the services the user wants to bind with the application. When deployed, we made sure that the applications are deployed on different execution environments called Droplet Execution Agents (DEA). Cloud Foundry has a Droplet Execution Agent (DEA) on each node in their grid and one DEA can have more than one applications deployed on it. With the help of the Cloud Foundry eclipse plug in, it is possible to manually ensure that the deployed applications go on different DEAs.

2) Methodology of collecting results: In order to collect the results we identify two categories of methods namely,*Control methods*-are methods that do not contain antipatterns. *Antipattern Initiating methods*-are methods that initiate a call constituting an antipattern. We chose the method *findVets()* as a control method. Note that the method *findVets()* is a Database Access method, i.e. it accesses data from the DB during its execution. We chose this method particularly as we wanted the control method to be as close to the methods initiating an antipattern sequence as possible otherwise no close comparison of the relative scaling of the same could have been made with methods that access the DB (because then one can argue that any difference in scaling is due to access to DB itself and not due to an antipattern). Similarly *call()* is the antipattern initiating method that initiates the chain of calls for the Circuitous Treasure Hunt Antipattern.

We monitored the execution time of these two types of methods in Local Deployment as well as Cloud and calculated the Speedup for each of them. As mentioned earlier in this subsection, we take execution time readings through two modes of communication: RMQ and WS calls, i.e., two set of experiments. For calculating speedup we use the following ratio of execution times locally and on cloud. Intuitively, speedup can be related to as a ratio of the rate at which a method (or a group of methods) can execute on Cloud Vs that on a Local deployment. As this rate is inversely proportional to the execution time [21] we use the following formula for speedup in these experiments. This formula holds under conditions of steady state execution.

$$Speedup = \frac{Exec\_Time\_Local}{Exec\_Time\_Cloud}$$

B. Empty Semi Trucks

1) Test Bed:

- *Local Deployment:* The test bed for local deployment consisted of two machines, hosting two different sub-applications for simulating the Empty Semi Trucks antipattern. The first machine holds the Client application

viz. 'spring-petclinic' merged with code for making requests for i^{th} record in the 'Vet' schema. The Server application holds the 'Vet' schema. The Rabbit MQ Server was on the second machine in the RMQ experiment case.

- *Cloud:* The test bed in the cloud consists of two DEAs, hosting the two different sub-applications. Again, while deploying the applications on Cloud Foundry, we made sure that the applications are deployed on different machines.

2) Methodology of collecting results: We utilize the method call() which initiates the Empty Semi Trucks antipattern. We consider both ways of communication via RPC i.e. call() that uses Rabbit MQ, and another version with call() that uses Web Service calls to communicate. The control method remains the same as in CTH antipattern implementation. We then measure the execution time of this method in Local Deployment as well as Cloud, calculate the Speedup with respect to the control method.

C. Results

Measurement of the execution times of different parts of the system was a critical part of calculating the various speedups and we made sure that we took enough samples for this calculation. For the same we took at least 12 sample readings for each data set (with more readings for data sets with high standard deviation).

There were 5 cases (data sets) which were studied and measured each for a local deployment as well as on Cloud deployment. These 5 data sets consist of execution times for:

1) *findVets()* method which is a control method that accesses database - represented as CM-DBA
2) *call()* method using Rabbit MQ for remote communication while initiating a Circuitous Treasure Hunt antipattern - represented as CTH-RMQ.
3) *call()* method using Web service calls for remote communication while initiating a Circuitous Treasure Hunt antipattern - represented as CTH-WS.
4) *call()* method using Rabbit MQ for remote communication while initiating an Empty Semi Truck antipattern - represented as EST-RMQ.
5) *call()* method using Web service calls for remote communication while initiating an Empty Semi Truck antipattern - represented as EST-WS.

In the interest of space, we show a set of observations for two measurements: For CM-DBA in Figure 9 and for EST-WS as in Figure 10. We performed similar measurements for other cases as well. The set of results that we got after aggregating the data points for all the six cases is shown in Figure 11. It is easy to note that the execution times for the control method reduces significantly when the application is migrated to cloud (arguably because of superior underlying execution infrastructure on the Cloud), while those for the methods involving an antipattern do not show such improvement. In fact, in three out of the four cases involving antipatterns we observe a slowdown which means that the execution time increases on cloud as compared to execution time on a local deployment. It is apparent from this that antipatterns significantly deteriorate performance of parts of the application which possess them.

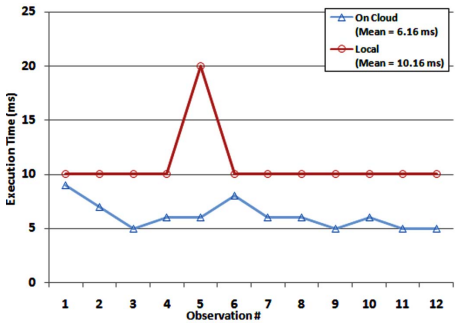

Fig. 9. Execution times for ***findVets()***-a control method with database access (CM-DBA)

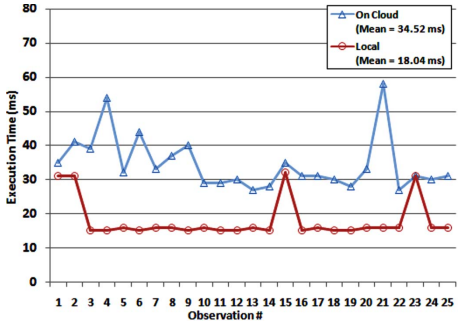

Fig. 10. Execution times for ***call()***-an EST initiating method using Web Service calls (EST-WS)

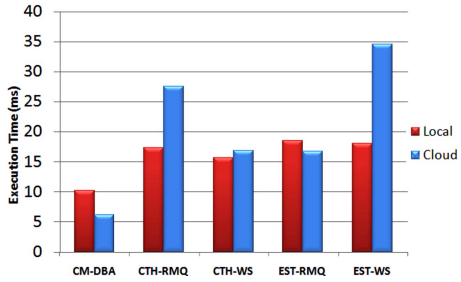

Fig. 11. Execution Time for different parts of the system

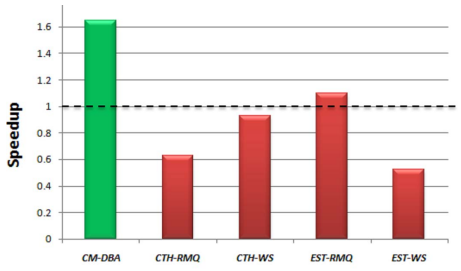

Fig. 12. Speedup for different parts of the system

The Figure 12 shows the comparative speedups of the various parts of the system. As can be observed here, the speedup of the control method was significantly more than the corresponding speedups of antipattern initiating methods. In fact for all but one of the antipattern initiating methods, the

speedups are lesser than 1, while that of the control method is more than 1.6 . Further, this trend is common across both the two antipatterns here. This highlights the detrimental effects of performance antipatterns if proper care is not taken and these get inadvertently introduced during a lift and shift Cloud deployment.

VI. Conclusion and Ongoing Work

Better performance is one of the major drivers of migrating to Cloud platforms. While Cloud computing promises elastic scaling capabilities, simply lifting and shifting a system to Cloud will more often than not be suboptimal. The inherent design of a software system dictates on how it will scale, and certain design antipatterns can potentially hamper software performance once deployed on Cloud. While there are certain studies on detecting design patterns from application code, there are no studies that detect performance antipatterns or show that these antipatterns can actually cause parts of an application to perform poorly when migrated to Cloud.

In this paper, we present our static analysis based approach to detect performance antipatterns in code. This approach is different from the (limited) existing work which depends on dynamic runtime information or significant human intervention in terms of annotated UML models to be able to detect these. Our approach utilizes deep analysis of the structure of an application as well as incorporates input on the expected deployments of the individual components of the same. We also presented an empirical evaluation which shows that parts of a software system where performance antipatterns exist, actually demonstrate poorer speed-ups (or even become slower) when migrated to the Cloud Foundry PaaS platform, as compared to other parts of the same application.

As our approach is based on static analysis, it suffers from producing more false positives than if we could use dynamic analysis. However, the overheads or feasibility of performing dynamic analysis in the Cloud migration assessment phase are huge - such a study will entail actually migrating the system to Cloud or simulate that migration to capture the needed data. Considering this, we believe that our approach is best suited to give quick actionable results without effort intensive setups. An experienced technical architect can then further vet our results in the particular Cloud context to remove some obvious false positives.

While we have not shown it explicitly, our approach can be iteratively used for a what-if analysis of different deployments of a system migration onto the Cloud. Certain antipatterns like Empty Semi Truck or Circuitous Treasure Hunt manifest themselves when the participating modules are deployed separately (across different domains/machines) and the tool can be used to evaluate the possibilities. This antipattern detection has been incorporated in a migration assessment tool [5] that we had built earlier and we have started to evaluate real life applications before Cloud migration. We are in the process of adding support for detecting more performance antipatterns as well as other design and architecture level patterns that become important when migrating to Cloud.

VII. Acknowledgments

The authors would like to thank Chethana Dinakar and Afsal Marattil for all their help in implementing parts of the Performance Antipatterns Detection (PAD) prototype.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2nd international workshop on Software and performance*, ser. WOSP '00. New York, NY, USA: ACM, 2000, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/350391.350420

[3] C. U. Smith, "New software performance antipatterns: More ways to shoot yourself," in *Proceedings of the Int. CMG conference*, 2002, pp. 667–674.

[4] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Proceedings of the Int. CMG conference*, 2003.

[5] V. S. Sharma, S. Sengupta, and S. Nagasamudram, "Mat: A migration assessment toolkit for paas clouds," *2013 IEEE Sixth International Conference on Cloud Computing*, 2013.

[6] V. S. Sharma and S. Anwer, "Detecting performance antipatterns before migrating to the cloud," *Poster/Short paper at the 2013 IEEE International Conference on Cloud Computing Technology and Science*, pp. 148–151, 2013. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2013.166

[7] VMWare, "Cloud foundry," http://www.cloudfoundry.com/.

[8] "Heroku cloud application platform," http://www.heroku.com/.

[9] "Apache tomcat clustering/session replication how-to," http://tomcat.apache.org/tomcat-6.0-doc/cluster-howto.html.

[10] A. Wierda, E. Dortmans, and L. J. Somers, "Detecting patterns in object-oriented source code - a case study," in *ICSOFT (SE)*, J. Filipe, B. Shishkov, and M. Helfert, Eds. INSTICC Press, 2007, pp. 13–24.

[11] H. Washizaki, K. Fukaya, A. Kubo, and Y. Fukazawa, "Detecting design patterns using source code of before applying design patterns," in *ACIS-ICIS*, 2009, pp. 933–938.

[12] J. Misra, K. M. Annervaz, V. S. Kaulgud, S. Sengupta, and G. Titus, "Java source-code clustering: unifying syntactic and semantic features," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012.

[13] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic design pattern detection," in *IWPC*, 2003, pp. 94–.

[14] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, ser. QoSA '12. New York, NY, USA: ACM, 2012, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2304696.2304704

[15] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–91, 2008.

[16] "Soot: a java optimization framework," http://www.sable.mcgill.ca/soot/.

[17] "Checkstyle 5.6," http://checkstyle.sourceforge.net/.

[18] "Henderson-sellers," http://eclipse-metrics.sourceforge.net/descriptions/pages/cohesion/HendersonSellers.html.

[19] "Lack of cohesion of methods (lcom and lcom hs (henderson-sellers))," http://codenforcer.com/lcom.aspx .

[20] "Springsource/spring-petclinic," https://github.com/SpringSource/spring-petclinic/.

[21] K. S. Trivedi, *Probability and statistics with reliability, queuing and computer science applications*, 2nd ed. Chichester, UK: John Wiley and Sons Ltd., 2002.