2013 IEEE International Conference on Cloud Computing Technology and Science

Detecting Performance Antipatterns before migrating to 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*—Performance is one of the key drivers for migrating existing systems to Cloud. While Cloud computing platforms come with a promise of scaling on demand, simple lift and shift of an existing application to Cloud would often not be the best solution. The design of a software system has a significant bearing on its performance and while migrating to Cloud, certain design patterns, can be detrimental to software performance. The area of detecting performance antipatterns automatically in context of Cloud migration and assessing their effects on performance is however unstudied. In this paper we present an approach to assess a system for known performance antipatterns, before Cloud migration. Our approach leverages static analysis and also factors in information about the prospective deployment on Cloud to evaluate whether certain antipatterns become prominent if the system is migrated to Cloud. We have found that the presence of these performance antipatterns can actually worsen the performance of parts of a software system containing them, when compared to those without these.**

***Keywords*—Cloud computing, Performance Antipatterns, PaaS**

I. Introduction

As more organizations adopt Cloud Computing for developing and deploying applications on outsourced infrastructure, there is a buzz of expectations around how migrating existing applications to Cloud equals high application performance. However, while Cloud Computing promises seemingly elastic capacity and scaling, applications being migrated to Cloud should also be able to exploit this elasticity and be inherently scalable. 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 simply migrating one's application to Cloud would result in high software performance is trivializing a much more complex problem.

The first aspect - the inherent architecture and design of the software system, is perhaps the most important of all. The design of a software system governs how it utilizes its environment and the underlying infrastructure. Thus while assessing migration of a software system 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 may manifest in a systems control flow, deployment, code, data flow, memory and 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 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.

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 those from the perspective of them becoming significant if the application is deployed on Cloud. To the best of our knowledge our approach is unique in that no other work has attempted to detect performance antipatterns in the context of assessing migration to Cloud. We have also empirically found that parts of the software system where we detect these antipatterns can cause performance bottlenecks on the Cloud.

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. We then very briefly describe some results from our early experiments which show the detrimental effects of performance antipatterns in Section 4. We then discuss some observations and conclude the paper in Section 5.

II. Performance Antipatterns and the Cloud

Cloud Computing's promise of elasticity and high performance is based on the on-demand horizontal scaling and load balancing capabilities of typical cloud platforms. However, this

978-0-7695-5095-4/13 $31.00 © 2013 IEEE
DOI 10.1109/CloudCom.2013.166

promise can easily be misinterpreted as "Putting my application on Cloud is guaranteed make it perform much better!". This is obviously not true and 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. 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. 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. 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

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. Others like Blob [2] will cause processing capabilities to be over utilized and thus would again require more powerful Cloud tiers. 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, their effects on performance and example manifestations. However there have been few automated approaches to find performance antipatterns (specially from code) and assessing the effects of such antipatterns in the Cloud computing context. Addressing these challenges form the crux of this paper.

There have been a few related works in detection of design patterns automatically or semi-automatically however none of those is in the context of migration to Cloud. In [5], the authors use a technique called Formal Concept Analysis, to detect frequently used structural design constructs without upfront knowledge. In [6], the authors 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. Recently, in [7], the authors convert the software architectural model of an application to a performance model and identify problematic patterns using First order Predicate Logic to represent antipatterns and use thresholds for various features like maximum connections, messages, utilization, etc.

The approach we take is targeted towards assessment of an application for presence of performance antipatterns *before* an actual migration and deployment on a Cloud. Our focus is only on performance antipatterns and the approach is based on static analysis of application code coupled with what-if analysis of the expected deployment of various parts of 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 [6] 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 exhibit performance antipatterns.

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.

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 [6] and we need 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, which is then a basis of annotation with possible deployments 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].

As shown in Figure 1, we use an AST Parser to parse code and find out all the method calls that are present within loops.

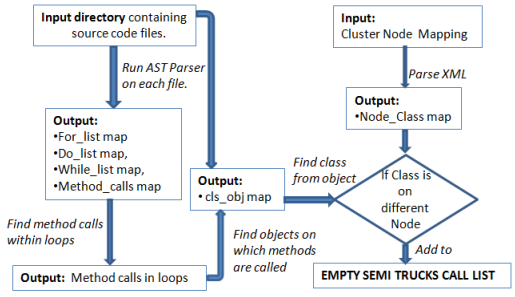

Fig. 1. Empty Semi Trucks Antipattern detection

We can not base our decision on the number of iterations in the loop because that is not always determinable by static analysis of code. We claim that such a coding practice is 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 these, 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 very detrimental. 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

Just like a game of *Treasure Hunt* where one has to move from one clue to another in search of hidden treasure, Circuitous Treasure Hunt 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 one by one, to the object that made the original call. [2]. As one would appreciate, this entails huge overheads in creating and destroying intermediate objects as well as communication across the network.

The target here is to find chain of methods such that one method calls another and so on subsequently (calls going across classes) 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 2 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 during the chain of method calls. We use Soot framework to create a Jimple representation of the code [8]. 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.

C. Blobs

A Blob as explained in [2] is a module "that performs most of the work of the system, relegating other classes to minor,

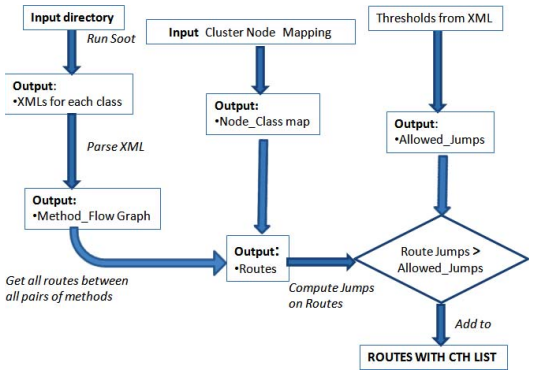

Fig. 2. Circuitous Treasure Hunt Antipattern detection

supporting roles". Though these can also be packages, for simplicity, we assume these typically are complex controller classes surrounded by simple classes that serve only as data containers. Data Container classes typically contain only getter and setter methods and perform little computation. 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 five 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 highlights classes that do much of the work. The fourth metric is *Lack of Cohesion in Methods* using Henderson Seller method (LCOMHS) which signifies whether or not the class has multiple responsibilities. High values of all these metrics tend to highlight a 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. Finally, *Access to Foreign Data* (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 if: (i) It has complexity equal to the number of methods it possesses i.e. It has getter and setter methods only, (ii) It is on a different node than the caller class.

For calculating the NOM, WMC, and CFOC metrics, we used Checkstyle [9] and JDT API. In Checkstyle's 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 can compute the LCOMHS, as defined in [10].

IV. Effect of Performance Antipatterns on Cloud

We set out to evaluate the adverse effects antipatterns have 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: CTH and 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. For making our experiments more generic, we created two versions of these distributed applications - one which utilizes Rabbit MQ (RMQ) and other that uses web service (WS) calls for communication. We then compared the scaling of execution time (speedup upon as-is deployment on Cloud) between control methods (i.e. methods without antipatterns) and the methods that manifest antipatterns to validate our hypothesis. We used Cloud Foundry [11] PaaS for our experiments.

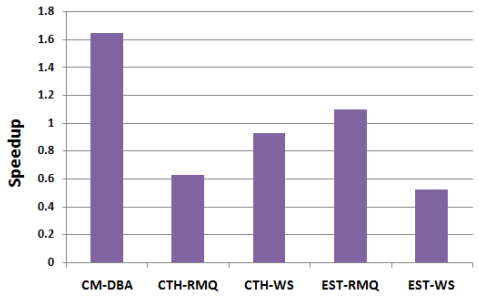

Fig. 3. Speedup for different parts of the system

The Figure 3 shows the comparative speedups of the various parts of the system. A speedup of less than 1 denotes slowing down of that part, and vice-versa. As can be observed here, the speedup of the control method (CM-DBA) was significantly more than the speedups of antipattern initiating methods. This trend is common across the two antipatterns types (CTH & EST). This highlights the detrimental effects of performance antipatterns if proper care is not taken during a lift and shift Cloud deployment and validates our claim that detecting these is essential while migrating to Cloud.

V. Conclusion and Ongoing Work

Better performance is one of the major drivers for migrating to the Cloud. While Cloud computing promises elasticity, simply lifting and shifting a system to Cloud will often prove to be suboptimal. The inherent design of a software dictates how it will scale, and certain design antipatterns can potentially hamper software performance once deployed on Cloud.

In this paper, we introduced a static analysis based approach to detect performance antipatterns. This approach is different from most existing works which depend on dynamic runtime information or human intervention.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 very briefly presented some results from an empirical evaluation which show 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.

We use a conservative static analysis approach because it is better to be critical about a system migration than to miss certain problem prone parts. The overheads 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 data. Hence, we believe that our approach is best suited to give quick actionable results.

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 across different domains/machines. If refactoring can't be done easily, one can use our analysis to identify deployments wherein the effect of the performance antipatterns is diminished. We intend to continue our experimentation and present our testbed, study and results, in more detail next.

This antipattern detection approach is currently being incorporated in a migration assessment tool [12] that we had built earlier and will be used to evaluate real life system 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. In our future work, we aim to work on making pertinent recommendations on how to mitigate or refactor components of application with antipatterns.

VI. Acknowledgments

Authors would like to thank Chethana Dinakar for her help in implementing parts of the Antipattern Detection 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] 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.

[6] 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.

[7] 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

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

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

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

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

[12] 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.