Glaucus: Predicting Computing-Intensive Program's Performance for Cloud Customers

Xia Liu[1,*], Zhigang Zhou[1], Xiaojiang Du[2], Hongli Zhang[1], and Junchao Wu[1]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China
baleitu315@gmail.com,
{zhouzhigang,zhanghongli,wujunchao2011}@pact518.hit.edu.cn
[2] Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA
dux@temple.edu

Abstract. As Cloud computing has gained much popularity recently, many organizations consider transmitting their large-scale computing-intensive programs to cloud. However, cloud service market is still in its infant stage. Many companies offer a variety of cloud computing services with different pricing schemes, while customers have the demand of "spending the least, gaining the most". It makes a challenge which cloud service provider is more suitable for their programs and how much computing resource should be purchased. To address this issue, in this paper, we present a performance prediction scheme for computing-intensive program on cloud. The basic idea is to map program into an abstract tree, and create a miniature version program, and insert checkpoints in head and tail for each computable independent unit, which record the beginning & end timestamp. Then we use the method of dynamic analysis, run the miniature version program on small data locally, and predict the whole program's cost on cloud. We find several features which have close relationship with program's performance, and through analyzing these features we can predict program's cost on the cloud. Our real-network experiments show that the scheme can achieve high prediction accuracy with low overhead.

Keywords: Cloud computing, predict, cost, performance.

1 Introduction

Cloud computing has recently became an absorbing platform that attracted a substantial amount of attentions from both industry and academia [1-5]. It is a promising computing paradigm which provides a great platform with its spectacular storage capacity and powerful computational capability. As the advantage list of cloud, customers have motivated to outsource their computing-intensive applications to cloud. However, customers have no idea about how to buy resources of the cloud. So it has become a crucial challenge for the development of the cloud. Due to interest conflict, customers do not believe the resource purchase strategy given by cloud, while customers lack of professional knowledge. And one of the things they care about most is

D.-S. Huang et al. (Eds.) : ICIC 2013, LNCS 7995, pp. 285–294, 2013.
© Springer-Verlag Berlin Heidelberg 2013

how to run more applications spending less money meanwhile observing SLA. To our best knowledge, the existing methods can't resolve this crucial problem.

Realizing the problem above, we aim to develop a middleware to predict the resource cost of the customers' application on cloud without migrating the application. Taking use of the middleware, customers can know the general status of running an application, and this will guide customers to buy the cloud.

To build such a middleware, we first need to analyze customer's applications and create the miniature version of the application. Specifically, we transform the application into abstract tree and then we separate it into many computable independent units and compare the similarity among them. Based on this, we create a miniature version program. We use the method of dynamic analysis. By running the miniature version program on small dataset locally, we predict the performance of application on large dataset on the cloud. And we find several features that have close relationship with the performance of application. The experimental results show that our middleware can predict the performance accurately. Our contributions are summarized as follows.

- Our middleware can predict customers' applications performance, and provide reference for customers to run more applications spending less money.
- We predict the performance of applications without migrating any application to the cloud.

2 Framework

As shown in Fig. 1, there are three kinds of entities in this system: cloud, customers, and middleware. We define our system formally first. To predict applications' performance, middleware need three parts of input from customers and the cloud respectively. The data needed to be transported to the middleware is defined as a triple $\mathrm{T}(D_A, D_T, f(D_T))$, where D_A refers to customer's application, D_T refers to the cloud type which customer wants to buy, and $f(D_T)$ refers to environmental difference between the cloud and local. After customer input cloud type D_T, middleware gets a five-tuple parameter $F(E_C, E_M, E_S, E_B, E_O)$ of the cloud, where E_C refers to CPU, E_M refers to memory size, E_S refers to storage space, , E_B refers to bandwidth and E_O refers to operating system. Based on the information above, middleware can calculate environmental difference $f(D_T)$. Based on this, our middleware predict the performance of applications and return the result to customers to give them suggestion.

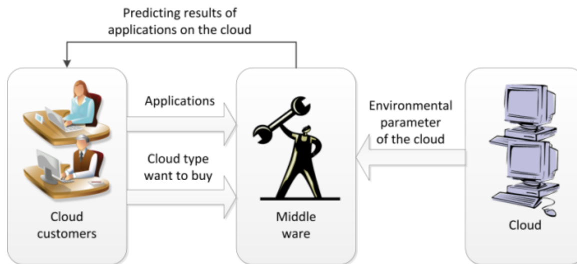


Fig. 1. Framework of middleware

3 Miniature Version Program

3.1 Abstract Tree

As our goal is to estimate customers' applications, we first need to analyze application. It is important to normalize application for the aim to analyze it easily, so we transform the application to abstract tree.

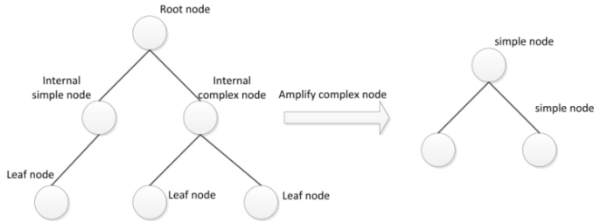


Fig. 2. Abstract tree

Table 1. Building abstract tree

Algorithm 1 Building Abstract Tree

INPUT:
Customer's Application: tokens
OUTPUT:
Abstrat Tree
MAIN
1: $n_{root} \leftarrow NULL, i = 0$
2: **while** $token \neq NULL$ **do**
3: **if** $token = "while"$ **then**
4: ***new*** $n_i = NULL, n_i = N_C$
5: $n_i.fathernode = n_{i-1}, i++$
6: **else if** $token = "for"$ **then**
7: …
8: **else if** $n_{i-1} = N_C$ & n_{i-1} *is completed*
&*token* ∈*simple token* **then**
9: ***new*** $n_i = NULL, n_i = N_S$
10: $n_i.fathernode = n_{i-1}, i++$
11: **else if** *token* ∈*simple token* **then**
12: *merge(token,* n_{i-1}*)*
13: …
14: **end if**
15: **end while**
16: **return** C

Here we give the definition of abstract tree. As Fig. 2 shows, there are three kinds of nodes in an abstract tree. Among which, root node N_R refers to the beginning of application, internal node N_I refers to different middle parts of application and leaf node N_L refers to end of application. Specifically, internal node includes two kinds of nodes which are called simple node N_S and complex node N_C. As we define, simple node refers to uncomplicated part of application (e.g., assignment statement), and complex node refers to complicated blocks in the application (e.g., loop, function call). At the same time, every complicated block also includes many simple nodes. That means we take every complicated block as a whole part “complex node”, meanwhile, the complicated block is also be analyzed and transformed to a sub-tree.

3.2 Independent Computable Unit

Based on abstract tree, we transform the whole program into a uniform structure. As our goal is to get miniature version program, we first split the main branch of abstract tree into independent computable units.

Definition 1. Independent Computable Unit: $U = \{u_i | u_i \in U,\ u_i \cap u_j = \emptyset\}$ Specifically, independent computable unit is a block of program which contains a series of successive statements, and there is no statement having relationship with this block outside it.

We find independent computable unit based on the abstract tree. First, we scan the abstract tree from bottom to top. If statements included in father node have semantic relation with the node which is being scanned, we merge father node with this node. Otherwise, father node belongs to a new independent computable unit.

3.3 Miniature Version Program

Although we get abstract tree, this branch is still too large to be analyzed, so we aim to get a miniature version of application.

Table 2. Typical Structures’ Running Time Analysis

Structure	Running time
While	$looptimes * \sum_{s_i \in while\&judge} get_time(s_i) + get_time(s_{judge})$
Switch	$get_time(s_{judge}) + \sum_{s_i \in switch} get_time(s_i)$
For	$looptimes * \sum_{s_i \in For\&judge} get_time(s_i) + get_time(s_{judge})$

We first separate main branch into several independent computable units, then reduce each unit. As different unit has different structure and also has different importance, if we want to get a more accurate miniature version program, we should not reduce each unit according to the unified standard. Therefore, we use the method of

analyzing running time ratio of different program structure, and then define the reducing ratio of each unit. As Table 2 illustrates, we analyze reducing ratio taking three typical structures.

Based on the definition of different structures' running time formula, we will figure out the whole structure's running time if we know each statement's running time. Then we split every statement into machine language. As Table 3 shows, we get the CPU cycles of machine instruct.

Table 3. The CPU Cycles Of Machine Instruct

Index	CPU cycles	Instruct
23/0	= 2	;AND r,r
23/1	= 6	;AND r,m
31/1	= 7	;XOR m,r
33/0	= 2	;XOR r,r
42	= 2	;INC eDX
28/0	= 2	;SUB r,r8
7C	= 7	;JL rel8

Based on the analysis above, we know CPU cycles that each statement need. So we define unit's reducing ratio as the inverse ratio of CPU cycles of each unit. Then we reduce each unit according to reducing ratio. Specifically, we reduce each unit's number of statements in terms of reducing ratio of the unit, then we get miniature version program. Formula (1) expresses the cost of the miniature version.

$$C_m = C_r * \varphi * \theta \tag{1}$$

So we enlarge cost results of program's cost as follows:

$$C_r = (C_m / \theta) * \varphi^{-1} \tag{2}$$

where C_r is real cost of program, C_m is miniature version program's cost, φ is complexity function of different structure and θ is reducing ratio of each unit.

4 Predicting Program's Performance

4.1 Core Variable Set

After getting the miniature version program, it is crucial to analyze it to find out the relationship between application and resources it have to occupy on the cloud.

As the complex node takes most of running time and also more other resources, we mainly analyze the complex node. Through analyzing complex node, we find out that in the complex node, what have close relationship with the resources the application occupies are some statements that contain the special variables. We define these variables as core variables, which are the variables which have high weight because they

appear in the miniature version program more frequently or appear in some important statements (e.g. malloc statement).

Definition 2. Core variable set: $V=\{v_i \mid v_i \in V, w_{v_i/w_{tree}} > \alpha, i \in [1,n] and 0 < n \leq \max_num(N_v)\}$, where v_i is one of the core variables which are in the miniature version program, w_{v_i} is weight of v_i, w_{tree} is total weight of miniature version program tree, α is the threshold which limit the standard weight of v_i, n is number of core variables and $max\ _num(N_v)$ is the total number of the variables appearing in miniature version program.

To find out core variables, we need to calculate weight of variables in the complex node. As described above, we have already known the CPU cycles each statement need, so we define statement's weight as corresponding CPU cycles. Then we calculate variable's weight according to the weight of statements which contain the variable. The weight of variable v_i is given by:

$$w_{v_i} = \sum w_{s_i} * n_{s_i} \tag{3}$$

where w_{s_i} is weight of statements which contain variable v_i and n_{s_i} is number of times these statements appears in the miniature version program.

After getting every variable's weight, we sort these variables and find out all of the variables whose weight are bigger than α. According to this method, we get m core variables.

4.2 Dynamic Analysis

As the performance of an application has great relationship with frequency of memory page swapping, we analyze several elements influencing memory swapping. We analyze two important features that are close related to the core variables, which are variables' frequency V_F, and average density V_D. Specifically, the frequency of core variables refers to the percentage of times a core variable occurring in the miniature version program, and average density refers to average distance of statements which contain a specific core variable.

Through the method of dynamic analysis, we find that when given fixed environmental parameters, performance of program will reach bottleneck as dataset becomes larger. So we analyze performance of program from two aspects. Before program's performance reaches bottleneck, we find that cost of program increases linearly approximately. We first run miniature version program on a specific small dataset, and get two pairs of program's cost (i.e. time cost and memory cost). Then predict cost of program before program's performance reaching bottleneck. The memory cost of program is given by:

$$m_s = \frac{M_2 - M_1}{D_2 - D_1} * d \tag{4}$$

where m_s refers to memory resource when running program on the specific small dataset, d is scale of dataset, M_1 and M_2 are memory size of program when running the program on two specific data scale respectively, and D_1 and D_2 are corresponding data scale.

As we know the maximal memory size M_{max} of our platform, we can figure out corresponding data scale when the program taking use of all the memory. So the largest data scale before program's performance reaching bottleneck is given by:

$$D_{\max} = \frac{M_{\max} * (D_2 - D_1)}{M_2 - M_1} \tag{5}$$

And by the same method, we can figure out running time of program:

$$t = \frac{T_2 - T_1}{D_2 - D_1} * d \tag{6}$$

where t_s refers to running time when running program on the specific small dataset, and T_1 and T_2 are running time of program when running the program on two specific data scale respectively.

Based on analysis above, we figure out the running time T_{max} of program when program's performance reaching bottleneck, which is given by:

$$T_{\max} = \frac{T_2 - T_1}{D_2 - D_1} * \frac{M_{\max} * (D_2 - D_1)}{M_2 - M_1} \tag{7}$$

Then we know the inflection point which represents the bottleneck of program's performance when given a fixed environmental parameter. After that, we predict performance of program based on the two important features. As we know, different features have different importance, we define the two features weight as w_{V_F}, and w_{V_D} respectively. And the running time of program on big dataset is given by:

$$t_b = \frac{T_2 - T_1}{D_2 - D_1} d + \left(w_{V_F} V_F + w_{V_D} V_D \right) T_{\max} \tag{8}$$

And memory resource that program occupy is given by:

$$m_b = \frac{M_2 - M_1}{D_2 - D_1} d + \left(w_{V_F} V_F + w_{V_D} V_D \right) M_{\max} \tag{9}$$

Based on our analysis of program's cost locally, we can figure out the whole program's performance on the cloud. We use the following performance model as our final performance model:

$$\Omega = \Upsilon \lambda \mu \tag{10}$$

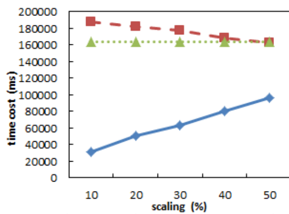
where Ω refers to the predicting value of program's cost on the cloud, Γ is the cost of miniature version program on our local middleware platform, λ is environmental parameter difference degree between local and cloud, and μ is the reducing ratio between real program and the miniature version program.

5 Evaluation

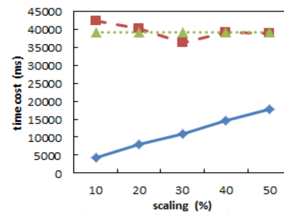
To evaluate the performance of our scheme, we run real computation-intensive application on both local platform and the real cloud platform. Based on this, we compare the results of application's cost of the two platforms. The configuration of our experiments is as follows:

Our local platform is a PC with Ubuntu 12.04 CPU, 512 MB Memory and 20GB hard disk. And we build our cloud test platform with 8-core 2.93 GHz Intel Xeon CPU, 24 GB memory, 500 GB and installed with Linux 2.6.18.

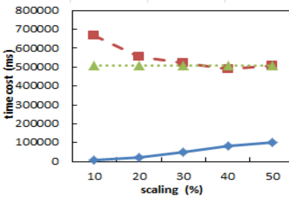
To find the reducing scale of miniature version program, we do experiment based on four typical computing-intensive programs (i.e. matrix multiplication, red-black tree, heap sort, maximal sum of submatrix). As Fig. 3 shows, when changing scale of program, the predicting of original program's performance also changes. And when reducing scale percentage is 20%, prediction result is the best.



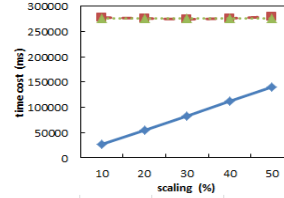
(a) matrix multiplication



(b) red-black tree



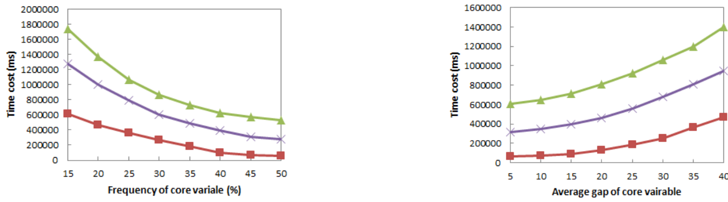
(c) heap sort



(d) maximal sum of submatrix

Fig. 3. Changing reducing scale of miniature version program

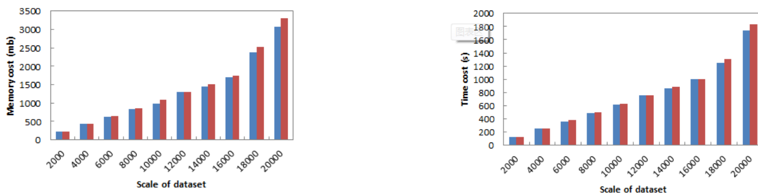
To prove the two features(i.e. frequency and average gap) have great influence on program's performance, we run our benchmark program locally, and change frequency and average gap of a specific core variable under the condition that the program filling all the available memory. And we find out that program's running time changes obviously with the variety of the two features. As Fig. 4(a) shows, program's time cost changes sharply as frequency of core variable changes from 15% to 50%. And Fig. 4(b) illustrates when changing average gap of a fixed core variable from 5 to 40, program's time cost also changes greatly.



(a) Changing frequency of core variable (b) Changing average gap of core variable

Fig. 4. Application's cost changes with features

Finally, we run our benchmark program on our local middleware platform with the dataset scale of 2000 and 4000 respectively. Then we predict cost of program on large dataset scale up to 20000 based on our scheme, and also run the program on cloud test platform. As Fig. 5 shows, our middleware can accurately predict the performance of computing-intensive applications on cloud with small dataset scale.



(a) Prediction result of program's memory cost (b) Prediction result of program's time cost

Fig. 5. Prediction result of program's cost

6 Related Work

Many efforts have been made to predict performance of the computing-intensive program. Li et al. [6, 7] compares performance of multiple cloud servers. This work is used to predict the performance of CPU-intensive applications across a large collection of CPU types and gives purchase suggestions for computing-intensive tasks and Web application. Zhang et al. [8] presents a performance prediction scheme that constructs a miniature version program to run in local machines, and then replays it in cloud to get the performance ratio between local and cloud. Ye et al. [9] proposes a testing algorithm based on HyperSentry [10] to detect SLA violations of physical memory size in virtual machine (VM). Sommers et al. [11] proposes a novel active measurement methodology to monitor whether the characteristics of measured network path are in compliance with performance targets specified in SLAs. Study [12] proposes a new passive traffic analysis method for on-line SLAs assessment, which reduces both the need for measuring QoS metrics as well as the interactions between the ingress and egress nodes in the network.

7 Conclusion

In this paper, we propose a method of predicting computing-intension program's performance on cloud. We identified and addressed three key challenges: (1) how to find unified form of computing-intensive applications for analyzing it easier, and (2) how to predict the performance of applications on large scale dataset without migrating any application to the cloud (3) how to provide reference for customers to run more applications spending less money. Our experiments showed that our scheme can achieve accurate prediction results with low overhead.

Acknowledgment. This work is supported by National Basic Research Program (973 Program) of China (2011CB302605), the project of National Natural Science Foundation of China (60903166, 61100188, 61173144), and National High Technology Research and Development Program (863 Program) of China (2011AA010705).

References

1. Amazon Web Service, `http://aws.amazon.com/`
2. Google AppEngine, `http://code.google.com/appengine/`
3. Armbrust, M., Fox, R.G.A., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A.: Stoica, 1., Zabaria, M.: Above the Clouds: A Berkeley View of Cloud Computing. University of California, Berkeley, Tech. Rep. (2009)
4. Sripanidkulchai, K., Sahu, S., Ruan, Y., Shaikh, A., Dorai, C.: Are Clouds Ready for Large Distributed Applications? In: Proc. SOSP LADIS Workshop (2009)
5. Microsoft Windows Azure, `http://www.microsoft.com/`
6. Li, A., Liu, X., Yang, X.W.: CloudCmp: Comparing Public Cloud Providers. In: USENIX/ACM Symposium on Networked Systems Design and Implementation (April 2011)
7. Li, A., Liu, X.: CloudCmp: Shopping for a Cloud Made Easy. In: 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud (2010)
8. Zhang, H.L., Li, P.P., Zhou, Z.G., Du, X.J., Zhang, W.Z.: A Performance Prediction Scheme for Computation-Intensive Applications on Cloud. In: Proc. of ICC 2013 (2013)
9. Ye, L., Zhang, H.L., Shi, J.T., Du, X.J.: Verifying Cloud Service Level Agreement. In: Proc. of Globecom 2012 (2012)
10. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In: Proc. of the CCS 2010, Chicago, Illinois, pp. 38–49 (2010)
11. Sommers, J., Barford, P., Duffield, N., Ron, A.: Multi-objective Monitoring for SLA Compliance. IEEE/ACM Transactions on Networking 18(2), 652–665 (2010)
12. Serral-Gracia, R., Yannuzzi, M., Labit, Y., Owezarski, P., Masip-Bruin, X.: An Efficient And Lightweight Method for Service Level Agreement Assessment. Computer Networks 54(17), 3144–3158 (2010)