Efficient Algorithm for Dynamic Inference of Virtual Machine Resource Needs

Netra Agrawal, Rohan Sumant, Chetan Pangam, Sai Lalitha Sree V
Department of Computer Engineering.
Santa Clara University
Santa Clara, USA

Abstract—Currently, reputed cloud deployment models for IaaS (such as AWS) require the user to provide a VM configuration. This is important in delivering a certain QoS guarantees to the end user. While this approach has worked pretty well with traditional deployment scenarios for the cloud, it is inadequate if the application's needs are volatile and ever changing. In such a case, the precise resource needs of a VM are hard to describe at any particular point in time and therefore it is difficult to simply predefine an initial VM configuration. We wish to come up with an algorithm which efficiently and continuously infers VM needs and meets them appropriately.

Keywords—Cloud Computing; Virtual Machine; Dynamic loading; Resource utilisation .

I. Introduction

The field of cloud computing has witnessed immense innovation over the past decade. Consequently, it has become of vital importance to both, cloud providers as well as end users to be able to streamline the managerial complexities of a cloud environment. Virtualization allows cloud providers to create multiple Virtual Machines (VM) on a single physical server. This improves the utilization of resources. The user must provide the configuration of the VM required for their application.

The cloud users pay for the statically configured VM size. But these charges may be unreasonable especially for applications which have variable load. There may be cases where the user pays for the underutilized resources they hold when the load is light and there maybe cases where there is performance degradation when the load is heavy. To avoid this, there must be VM migration based on the overutilization or underutilization of the VM. We wish to come up with an algorithm which continuously and efficiently infers the VM needs and meets them appropriately.

II. Contribution

- Algorithm to determine the Virtual Machine resource needs.
- Dynamically check the CPU consumption.

III. Motivation

Presently, users have to specify their needs for CPU, bandwidth, memory, disk and UI at the time of request. It is difficult to figure out the size of VM suitable for the applications.

Cloud computing works on pay-as-you-use principle. It makes on-demand usage of resources like storage, bandwidth and computational power. Load is always changing it is not always constant. Users are charged on the basis of their initial request for the resources. Because of static VM configuration users have to pay for the resources they hold even when the load is light on the other hand if the load is heavy then there are chances of performance degradation.

So, for minimizing the performance degradation VMs are moved from overloaded to underutilized host. There are many overloaded and underutilised hosts. So, for VM migration it becomes necessary to classify hosts as underutilised and overloaded. For this purpose some detection algorithms are needed to dynamically classify the active hosts. Dynamic algorithms thereafter determine the optimal plans for the VM migration. They decide on when and how to allocate VMs during runtime. The problem is the time required by the VM to migrate is potential high. The time VM migration takes depends on the platform used.

VM migration policy reduces the violation of Service Level Agreement(SLA) as it helps in migrating VMs from the host before the host get overloaded. With the help of live VM migration,according to resources required VMs can be combined to minimal number of physical nodes.

According to research, high energy consumption may cause due to inefficient usage of resources. Many times servers are not operated to their full capacity which leads to

GitHub Link: https://github.com/COEN241Cloud9/Cloud9_public.git
Docker Image: rohansumant/ccproject2

extra cost of acquisition.It means keeping servers underutilised is inefficient from energy consumption view.Therefore by switching the idle nodes to low power modes like sleep or hibernation,reduction in energy consumption can be achieved and idle power consumption can be eliminated.

IV. Discussion and Related Work

In past decade, studies found that hardware's power inefficiency and quantity of computing resources are not only the reason for high energy consumption but also the inefficient utilization of resources. In most cases, servers being used 10% to 50% of their full ability, which Unnecessarily leads to extra cost of possession.[5] Another problem is even idle server required 70% of its peak time power requirement.[6]. Hence, underutilized servers are highly inefficient.

There are many techniques which can improve energy efficiency like energy efficient hardware, terminal servers, and thin clients. In cloud computing, virtualization technology is good solution for the energy inefficient problem. We can reduce the energy consumption by making idle nodes sleep or to switch to hibernate mode. Moreover, VMs can be adaptively consolidated to minimal physical nodes as per current requirement of resources by doing live migration.

However, Virtualization has some problems. Cloud computing environment should able to deliver the quality of Service (QoS) as defined in the service level agreements (SLA). Some applications have highly variable workloads which apparently cause dynamic resources usage. It can degrade the performance of VMs when there is increase in workload resulting in over resources usage. It may violate the SLA by poor performance, more response time or time outs. Over availability can help to ensure SLA, but there will be inefficient use of resources when workload is less. The better solution is to adjust the resources provisioning as per actual demand at particular time of the application.

To map VMs to physical servers is one of the NP-hard problem.There is a large amount of literature on this problem and many heuristics have been proposed. Simple static VM migration is not sufficient, as we know the fact that in some application, demand of resources is rarely constant. We will primarily focus on adaptive solutions.

A. Reactive Overload Detection

We will discuss some heuristics of detection of adaptive overload of hosts which are integrated in CloudSim framework and used as reference solutions. They analyzed current and recent VM behaviour. They use statistical data of utilization and putting some threshold value, decided whether host is overloaded or not. To know more details on these approaches, refer to [7].

A simple approach is to have a static threshold value (Tu). If the current static analysis of CPU utilization exceeds Tu then the host is considered to be overloaded. The threshold value Tu is typically set to 80 to 90%. There are some techniques used for static analysis to take decision that host is overloaded or not. Local regression (LR), Median absolute deviation (MAD), Interquartile range (IQR) are some of them.

Beloglazov et al. suggest adaptive threshold based heuristics which control the migration process of VM. But the main disadvantage of these techniques is unnecessary VM migrations due to not able to forecast the future utilization of host as its try to find out better solution for current state. Now we will discuss Proactive approaches.

B. Proactive Load Balancing Algorithms

Bobroff et al. have looked into the problem of migration of server as per demand and figure it out by forecasting the future demand which used to remap VMs to host regularly. This technique reduces the SLA violation as compared to previous count of static analysis techniques.

Kusic et al. also proposed limited lookahead control approach using Kalman filter. But their approach was taking half an hour for 15 nodes optimization. So the proposed approach is not possible in real world environment. In recent days, Kalyvianaki et al. improvised Kalman filter. It gives feedback to filter to allocate resources to VMs dynamically.

To regulates the utilization of a single tier virtualized server, Xu et al. designed and implemented a predictive controller. It consist of 3 forecasting algorithms-

1. ANOVA decomposition
2. Autoregressive model
3. Multipulse model

The was giving appropriate results for regular patterns but it fails for the new pattern given as input.

V. Problem Analysis and Proposed Solution

The central problem that we address in this paper revolves around the case where the consumption of virtual resources is not on par with the demands of the application. In this paper we limit the scope of the problem to CPU consumption. It must be noted, however, that virtual resources in the cloud environment are independent of each other. Therefore the solution proposed in this paper can be just easily extended to other aspects of a VM.

Let us denote a threshold function *fn* which determines whether the current CPU consumption percentage meets the QoS parameters as specified in the SLA. Therefore *fn* implies a bijection from a number no greater than hundred to {0,1}. It is important to note that the threshold function doesn't yield

any information about the extent to which the current CPU configuration is adequate or otherwise.

The algorithm we propose is the following: let us start with an initial VM configuration where the CPU prowess is unknown. Essentially, this means that we are deploying our application on a VM with random CPU configuration. We deploy our application on the said machine and check whether the threshold function holds, ie, whether the value returned is 1 (true). If yes then this implies that the current CPU prowess either meets or exceeds the current application's demands. In this case, we try migrate the application on a VM whose computational power is half that of the current configuration. We continue doing so until we encounter a machine on which the threshold function return a 0. Then, at last, we have a lower bound on the computational power our application needs. Then we reiterate the process within the search space between the lower limit and the last known valid configuration (ie, where the threshold function returned 1). The procedure to determine an upper bound on target CPU power is identical.

Pseudo code is as follows:

Input: An array of CPU configuration of host machines.

Output: The index of the machine whose CPU prowess is appropriate for the given application.

Algorithm 1: Pseudo Code for Dynamic Inference of VM resource needs

```
1:  procedure ccalgo
2: CPUconfigArray[] = Array of CPU configuration
3: currIndex = Randomly select one CPU configuration index
4: index ← go(0, length(CPUconfigArray)-1, currIndex);
5: if index > length(CPUconfigArray)
6:         index = length(CPUconfigArray)-1;
7: end if
8: return CPUconfigArray[index];

9: procedure go(left, right, currIndex)
10:   if left == right then
11:         return left
12:  end if
13: if left == right-1 then
14:         return right
15: end if
16: if check(currIndex, CPUconfigArray[]) then
17:         i=0;
18:         while currIndex+(1<<i)<=r && check (
              currIndex+(1<<i),CPUconfigArray[]) )
19:                 i=i+1;
20:         upperBound = currIndex+(1<<i);
21:         if i==0 then
22:                 lowerBound = currIndex;
23:         else
24:                 lowerBound = cuurIndex+(1<<(i-1));
25:                 end if
26:         if upperBound > right then
27:                 upperbound = right
28:         return
go(lowerBound+1,upperBound,lowerBound+1)
29:                 end if
30:     else
31:         i = 0
32:         while currIndex-(1<<i) >= left && (
          !check(currIndex - (1<<i), CPUconfigArray[]))
33:                 i = i+ 1;
34:         lowerBound = currIndex-(1<<i)+1
35:         if i == 0 then
36:             upperBound = currIndex
37:         else
38:             upperBound = currIndex-(1<<(i-1))
39:         end if
40:         if lowerBound < left then
41:             lowerbound = left
42:         return go(lowerBound,upperBound,lowerBound+1)
43:         end if
44:     end if

45: procedure check(currIndex,CPUconfigArray[])
46: if application runs properly
47:         thresholdFunction fn =1;
48: else
49:         thresholdFunction fn =0;
50: end if
51: return (!fn)
```

Example:

Assume that we have an array,

arr[CONFIG_SIZE] = {2,3,5,7,12,14,20,42,53}

Here the integers in the braces represent CPU instruction processing capability in some unit, say MIPS. Note that the input array is sorted. The algorithm proceeds as follows.

1) The defined range is from index 0, to index 8 (assuming that the array indexing is 0 based).
2) We start at a random index which lies within the given range, say 4.
3) The performance of the application is checked at index 4. If it is adequate (as inferred from the value returned by the threshold function) then indices on the left are scanned in an exponential fashion to determine the index which delivers inadequate performance. Therefore we check on indices, $4\text{-}2^0 = 3$, $4\text{-}2^1=2$, $4\text{-}2^2=0$. Had the threshold function returned 0 (ie inadequate), we would have applied the same procedure to the right side of the current range.
4) Note that after each repetition of the previous step, we are shortening the range under consideration. Eventually, the range [l,r] degenerates to the trivial condition where r = l+1.

In essence, the problem has been reduced to finding the first point of transition in a monotonic sequence consisting only of zero and one. The challenge, however, lies within the way cloud infrastructure is organized. Cloud environments are highly autonomous and are built around the principle of avoiding single points of failure. Therefore it is not possible for this algorithm to simply perform a binary search on candidate configurations, because in doing so the algorithm will start every time on just one set of machines (since a binary search tree has exactly one root). Our algorithm addresses this challenge by accommodating multiple points of entry.

On closer analysis of the problem we realize that the algorithm itself is a slightly modified version of the Galloping search algorithm [8] as first put forth by Jon Bentley and Andrew Chi-Chih Yao in 1976. The recurrence for our algorithm is:

$$T(n) = T(n/2) + O(logn)$$

It can be derived that the upper bound for this algorithm does not exceed $(logn)^2$. Notice that since we customize the second part of the original algorithm the execution remains stateless. This is a deliberate effort to ensure that the implementation is scalable enough to be adopted into mainstream deployment scenarios. It also ensures that the algorithm can be applied independent of the VM state therefore allowing it to be repeated at opportune moments where the possible cost of VM migration is less than the overhead of running an application on a VM ill fitted to its needs.

VI. Implementation and Evaluation

For implementation purposes we intended to deploy a compute heavy application on an actual cloud IaaS offering. Our intention behind this was to ascertain as precisely as possible the real CPU performance obtained on real world servers. The CPU consumption can be monitored and logged by linux utility commands such as *top, time.* Thus we can create an array of CPU profiles of candidate VM configurations. Once obtained, we can iterate over the values in the manner described above.

For evaluation purposes, we can run an application on a VM profile which is favoured most prominently by the clients of a particular platform and run another instance of the same application whose VM placement is guided by our algorithm. We then simply check which application goes to completion quickly, while also noting the VM migration costs our algorithm faces.

However, the problem we faced was that we could not get VM with different configurations to implement the project. Only one configuration on AWS is free. Thus, we have implemented a simulation for the project by using Docker Containers. We have created a Docker Image and limited the CPU share the image can use by using --cpus flag while running the container.

We have chosen Primality testing as a candidate application to be orchestrated by our algorithm and the reason is twofold. Firstly, it is a relevant problem which has been studied extensively with implications in fields ranging from bioengineering to cryptography. The industry standard for encryption, the RSA algorithm, makes use of the knowledge of a primality about a number. Hashing techniques used in peer to peer storage and distributed systems rely on large prime numbers so as to come up with robust implementations. Hence, determining the primality of a number is an important application.

Secondly, the algorithm proceeds in an iterative fashion. In each iteration, a random number (less than the number which is being checked for primality) is selected, which, based on Fermat's little theorem determines whether or not the number under consideration is a prime. Note that the test is actually for *compositeness* of the number, in that it determines with absolute certainty if a number is composite. Therefore, if after several iterations the result is false for each case, the chances that the number is a prime are very high. Such numbers are therefore termed as *probable primes.* Since this algorithm

proceeds in an iterative fashion, it is easy for us to determine whether or not the cpu capacity of the current machine is adequate or not. The threshold function can simply be defined as,

$$Threshold\,fn = \frac{iterations\ performed\ per\ unit\ time}{iterations\ required}$$

If this function yields 0, then the current processing power is inadequate.

Pseudo code of the program is as follows:

Input: Threshold value, sample size

Output: 1 if current processing power is adequate else 0

Algorithm 2: Pseudo Code for Primality Testing

```
1: clock_t t1,t2
2: required_cnt=stoi(string(argv[1])) = minimum numbers to
                    be checked. It is passed as
                    argument
3: N=stoi(string(argv[2])) = constant total numbers passed as
                    second argument
4: for i=0 to <N
5:    repeat
6:        t2=clock();
7:        diff=(t2-t1)/CLOCKS_PER_SEC= difference
                                    between clock time.
8:        if diff >3 then
9:                if i>required_cnt
10:                       cout<<1
12:               else cout<<0
13:               end if
14:       end if
15:       i++
16: end for
17: if flt_test(x)==true then
18:       print(prime)
19: else
20:       print(not prime)
21: end if

22: procedure flt_test(num,depth=20)

23:if num<=20 then
24:       cout<<"Number too small"
25: end if
26: while text_set.size()!=depth
27:       test_set.insert(rand()%num)
28: for x:text_set
29:   repeat
30:       if modpow(x,num,num)!=x then
31:               return 0
32:       end if
33: end for
34: return 1

35: procedure modpow(x,p,MOD)

36:. if p==0 then
37:       return 1%MOD
38: else if p==1 then
39:       return x%MOD
40: end if
41: sqr=(x*x)%MOD
42: if p%2==0 then
43:       return modpow(sqr,p/2,MOD)%MOD
44: else
45:       return (x*(modpow(sqr,p/2,MOD)%MOD))%MOD
46: end if
```

We have implemented the algorithm by running a python program *ccalgo.py*. In this program, we run the Docker image by using docker run command. Here, we are limiting the CPU share by using -cpus attribute in the run command.

docker run --cpus"0.20" ccproject2 /project/driver.sh

driver.sh is the shell script which will run the Input text file and the primality testing program. The Input text file contains randomly generated numbers which are of at most 9 digits long. The inputs to the Primality testing program are the threshold and the sample size. The threshold we have used is 350000, i.e at least 350000 numbers must be checked if they are prime or not within the defined time span.

VII. Results and Evaluation

In our simulation, the algorithm satisfactorily provides a runtime environment which is capable of running the application at the desired speed of progress. Compared to a naive deployment scenario which takes 53s to complete, the configuration selected by the algorithm is capable of completing the given task in 16s. Note that the best runtime

that can be achieved is of 7s, but such CPU capability is more than what the client expects.

Assuming that the total runtime of a program is proportional to the the total number of instructions to be executed, the below graph denotes the performance characteristics we would expect upon application of our algorithm. The gap on the Y axis, is an upper bound on the overhead any application would experience when orchestrated by our algorithm.

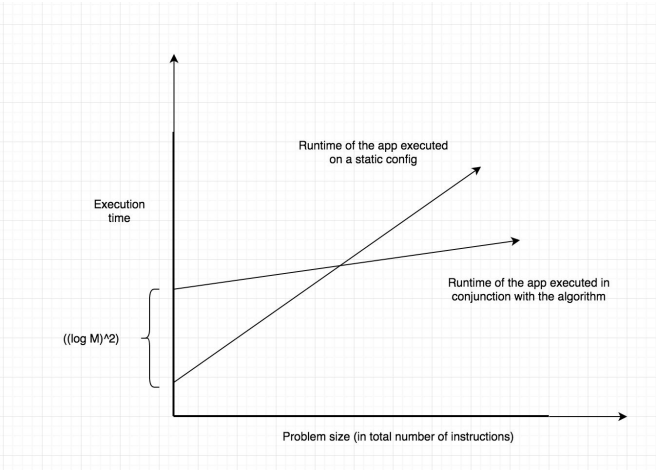

Figure 1. Theoretical projection of the performance of our algorithm.

VIII. Scope AND Limitations

The scope of the algorithm is limited only to CPU consumption. But the solution can be extended to other resources as the virtual resources in the cloud environment are independent of each other. The limitation of this algorithm is that all applications cannot be eligible candidates for this algorithm. Some applications may have needs that may change too quickly. Determining the appropriate configuration match for such applications becomes too expensive.

The kind of migration techniques being applied and their frequency also play a big role in ensuring whether this technique pays off. A compute heavy application usually also has a reasonably big data dependency graph. That is precisely the reason why scaling out is not an option unless you can tolerate for a major architectural overhaul of the application. In such scenarios, it makes sense for the application to be shifted onto an appropriately capable machine provided that the cost of transfer along with the new computational prowess ensures higher throughput than what the prior system would have yielded.

IX. Conclusion and Future Work

Thus, we designed an algorithm that can efficiently and dynamically infer the VM needs and meets them efficiently. Compared to a naive deployment scenario which takes 53s to complete, the configuration selected by the algorithm is capable of completing the given task in 16s. For the future work, the same algorithm could be applied to containers. Even though the simulation was done using Docker Container, the proposed algorithm was intended for Virtual Machines. As all the applications cannot be eligible candidates for this algorithm as mentioned in the Limitations, there can a program designed to determine whether or not an application can be optimized this way. The asymptotics of the algorithm could be improved significantly, by a careful study of the cloud provider's organization of infrastructure.

X. CHALLENGES FACED DURING THE PROJECT

- Could not get VM of different configurations to implement the algorithm as it was not economical.
- On AWS, only one instance was free.
- Tried another Simulator QEMU, but the problem with that was we could not access it through the terminal and even though it provided hardware simulation, it uses the local OS configurations.
- The program gives some errors if the index goes beyond the array limit.
- Even though the program has given constant results most of them time, there were few cases when we got varied results.

Acknowledgement

We thank Abhishek Gupta, Professor of Cloud Computing COEN 241, for the support and guidance throughout the project. We thank the members of other groups for their cooperation during the presentation of the project.

References

[1] S. Matthias, K. Michael, T. Sven, "*Predictive Load Balancing in Cloud Computing Environments Based on Ensemble Forecasting",* Retrieved from http://ieeexplore.ieee.org.libproxy.scu.edu/document/7573156/authors

[2] H. Rongdong ,J. Jingfei, L. Guangming and W. Lixin, "*Efficient Resources Provisioning Based on Load Forecasting in Cloud"*, Retreived from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3951090/

[3] Chin-Fu Kuo, Ting-Hsi Yeh, Yung-Feng Lu, Bao-Rong Chang, "*Efficient Allocation Algorithm for Virtual Machines in Cloud Computing Systems",* Retrieved from http://dl.acm.org/citation.cfm?id=2818878&dl=ACM&coll=DL&CFID=892172946&CFTOKEN=337466

[4] A. Beloglazov, "Energy-efficient management of virtual machines in

data centers for cloud computing," Ph.D. dissertation, The University of Melbourne, 2013.

[5] 1. Barroso LA, Hölzle U. The case for energy-proportional computing. *Computer*. 2007;40(12):33–37.

[6] 2. Fan X, Weber W, Barroso L. Power provisioning for a arehousesized computer. *ACM SIGARCH Computer Architecture News*.

[7] R. C. Chiang, J. Hwang, H. H. Huang, and T. Wood, "Matrix: achieving predictable virtual machine performance in the clouds," in 11th Int.Conference on Autonomic Computing, ICAC., 2014, pp. 45–56.

[8] Baeza-Yates, Ricardo; Salinger, Alejandro (2010), "Fast intersection algorithms for sorted sequences", in Elomaa, Tapio; Mannila, Heikki; Orponen, Pekka, *Algorithms and Applications: Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, **6060**, Springer, pp. 45–61, doi:10.1007/978-3-642-12476-1_3, ISBN 9783642124754.