UNIVERSITY OF OREGON

Reducing Cloud Reconfigurations via Co-locating Compensative Workloads and Services

by

Yueqi Zhu

A thesis submitted in partial fulfillment for the
degree of Bachelor of Science

in the
Department of Computer and Information Science
University of Oregon

June 2017

Abstract

Cloud computing, as a computing method based on Internet, allows varieties of shared hardware and software resources being provided through terminals or devices. Like all other Internet-based services, performing tasks in the cloud could consume resources. Thus using resources effectively is an important subject for the cloud service provider. In this thesis, we classify two cases of cloud environment and propose several algorithms to reduce cloud configuration cost based on them. For each algorithm we analyze the performance of it in detail and justify it by real world data.

Acknowledgements

First of all, I'm willing to thank my supervisor, Professor Lei Jiao. He offered me the opportunity to get into this research project and always tried his best to help whenever I met difficulty, from which I have gained a lot of knowledge and research skill. I also appreciate Dr. Jesus Omana Iglesias from Nokia Bell Labs, who also helped me a lot in this project.

Secondly I want to thank all faculties in the department of computer and information science in University of Oregon. They created a favorable environment to study and support me to get my degree here.

At last I would like to thank my friends and family. They are all important roles in my life. It would be impossible for me to get so far without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In cloud environment, cloud service providers are responsible for providing varieties of cloud service, such as computing and storage, to their customers. For example, in the SaaS (Software as a Service) model, customers should be able to access the software and data though Internet, and cloud service provider should maintain such a platform and make sure it is working correctly. Cloud services are often supported by a large network data center. In such an environment, resources held by the service provider are always limited. Thus it is important to reasonably allocate resources to customers.

There are numbers of services distributed across the environment, where tasks are assigned to. Each service has a certain amount of VMs (virtual machines) to perform tasks. The number of VMs required could be the scale to weigh how large the workload of a task. As the time goes, the size of workload varies. So we need to adjust services by booting up or shutting down some VMs as the change of workload. A larger workload always requires booting up more VMs to complete the task, which could accordingly cost more time and energy. However, it is possible to save reconfiguration cost, according to the type of services. Here we are going to discuss the method to save reconfiguration cost, based on two different cases.

In homogeneous services, all services are of the same type and same scale of workloads, which means we can mix them to reach our purpose. Specifically, if a service which requires more VMs (which means some VMs in this service need to boot up) is mixed with a service which requires less VM (which means some VMs in this service need to shut down), we can reuse VMs which ought to shut down to reduce the number of VMs which is needed to boot up, and reconfiguration cost is saved by that. We will describe the method in detail for homogeneous case in Chapter 2.

In heterogeneous service, the type of services are not necessarily the same. Thus we need some specific parameters to measure workloads. Besides that, the method of mixing services is not applicable anymore since we cannot mix services with different type. In place of that, we transfer files which required by the task through services. As a result, we avoid booting up more VMs since we moved some task to spare VMs, and reconfiguration cost is saved. The method of heterogeneous case is described in Chapter 3.

Chapter 2

Homogeneous services

2.1 Problem formulation

We can treat each service as a vector $w[i]$, the entries in the vector represents the workload at corresponding time slot. We define the reconfiguration cost of a vector as the sum of the difference between peak value and vale value of all increasing phase (since we don't care about the cost of shutting down a VM).

In the case of homogeneous services, all services are of the same type. So we can mix vectors in order to save reconfiguration cost. When mixing two vectors, we add each entry of two vectors separately, for example:

$w[1] = [2, 9, 5, 5, 6]$, reconfiguration cost$= (9 - 2) + (6 - 5) = 8$

$w[2] = [9, 5, 8, 3, 9]$, reconfiguration cost$= (8 - 5) + (9 - 3) = 9$

The total reconfiguration cost of $w[1]$ and $w[2]$ is $8 + 9 = 17$

If we mix $w[1]$ and $w[2]$, we get a mixed vector $[11, 14, 13, 8, 15]$ with the reconfiguration cost of $(14 - 11) + (15 - 8) = 10$.

So we saved $17 - 10 = 7$ unit of workload by mixing $w[1]$ and $w[2]$.

Notation	Explanation
w	Set of vectors(services)
n	Number of vectors in w
l	Length of vectors
c	Server capacity

Table 2.1: Notation summary

There is another parameter called server capacity, which prevents us from mixing vectors unrestrainedly. With capacity c, each mixed vector cannot have any entry that is greater than c.

Our goal is saving reconfiguration cost as much as possible by mixing vectors without exceeding capacity.

2.2 Algorithm description

To have a comparison, we implement three methods on this problem: brute force, dynamic programing and greedy.

2.2.1 Brute force

We first traverse the number of mixed servers (groups), on each traversal, we use $M[i]$ to mark the location of the ith vector (i.e. $M[i] = l$ means the ith vector is in the lth group) and have two constrains:

1. $M[i] \leq m$ for all i, where m is the number of groups
2. $Max_t \sum_i w[i][t] \leq c$

The first constrain means the total number of groups cannot be greater than m, the second means that combined vector in each group cannot exceed the capacity. If both constrains stand, we calculate the total cost saving and find the maximum saving by that. Otherwise, we look at the next possible M (next possible combination), here if we can consider M as a m-base number and increase M by 1. This method is guaranteed to give us the optimal solution since it traverses all possible combinations for all possible number of groups. However, since there are n^n different M, it will take exponential time.

2.2.2 Dynamic Programing

We derive the recursive relationship so that one can easily implement either a recursive computation or a dynamic programming method to obtain the optimal solution to our workload categorization problem for a single server. Using $f(i, c)$ to denote the optimal reconfiguration cost saving for the first i workloads, and using c to denote the residual capacity (i.e., the free capacity or the capacity to be occupied), we consider the $(i +$

Algorithm 1 Brute force for homogeneous services

```
max_saving ← 0
for m = 1 → n do
  M ← [1] * n
  while True do
    exceed ← False
    initiate G [G is a list of groups, each group contains a mixed vector, which is all
    0 at first]
    for i = 1 → n do
      Add w[i] into the M[i]th group
      if max{G[M[i]]} > c then
        exceed ← True
        break
      end if
    end for
    if not exceed then
      mixed_saving ← Σ_{i=1}^{m} saving(G[i])
      if mixed_saving > max_saving then
        max_saving ← mixed_saving
      end if
    end if
    increase M by 1
    if M > [m] * n then
      break
    end if
  end while
end for

return max_saving
```

1)th workload $w[i]+1$, if we reject it to be part of the group, then we have $f(i+1,c)=f(i,c)$; if we accept it to the group, then we have $f(i+1, c-max\{w[i]+1\}) = reconfig(f(i,c), w[i]+1)$, where $c-max\{w[i]+1\}$ approximately calculates the new residual capacity, and $reconfig(f(i,c), w[i]+1)$ is a function that calculates the new reconfiguration cost saving, given the current reconfiguration cost saving $f(i,c)$. To sum up, we have the following:

$$f(i,c) = \begin{cases} max(f(i-1,c), reconfig(f(i-1, c-w[i]), w[i]))) & \text{if} w[i] \leq c \\ f(i-1,c) & \text{otherwise} \end{cases}$$

Note that, in the above, one vector subtracting another vector is defined as using each element of the former to subtract each corresponding element of the latter, and one vector being no larger than another vector is defined as each element of the former being no larger than each corresponding element of the latter. The function

$reconfig(f(i,c), w[i]+1)$ to calculate the reconfiguration cost saving of mixing the existing workloads, i.e., a subset of $\{w[1], w[2], , w[i]\}$, and the workload $w[i+1]$ based on the previous models and definitions. Note that since we fill each group one by one, the order of vectors would influence our result. Therefore, we can randomize the order of vectors and run this algorithm on them for several times to obtain a better solution.

Algorithm 2 Dynamic programing for homogeneous services

```
mixed_saving ← 0
while True do
  for i = 1 → n do
    initiate f as all 0
    if w[i] already in another group then
      continue
    end if
    if current group is empty then
      add w[i] into current group
      continue
    end if
    for j = c → 0 do
      if j ≥ max{w[i]} then
        if reconfig(f[i−1][j−w[i]], w[i]) > max(f[i−1][j]) then
          f[i−1][j] = reconfig(f[i−1][j−w[i]], w[i])
          add w[i] into current group
        else
          f[i][j] = f[i−1][j]
        end if
      else
        f[i][j] = f[i−1][j]
      end if
    end for
  end for
  mixed_saving = mixed_saving + f[n][c]
end while

return mixed_saving
```

2.2.3 Greedy

We also propose a greedy method that may not obtain the optimal solution like the above recursion, but can obtain a solution of reasonably good quality in faster execution. We sort all the workloads in the ascending order based on $max_{t\in T}(w[i][t])$ which can be deemed as the size of the workload $w[i]$. Note $max_{t\in T}(\sum_{i\in I} w[i][t] \leq \sum_{i\in I} max_{t\in T}(w[i][t]))$, indicating if the right-hand side of this inequality does not exceed the server capacity, the left-hand side is guaranteed not to exceed it. After doing the sorting, we traverse the workload from the one with the smallest size to the one with

the largest size. We select a workload to be part of a group in order. And if selecting a vector will exceed the capacity of a group, we create a new empty group and place the vector in it.

Algorithm 3 Greedy for homogeneous services

```
Sort all vectors by their peak value (increasing order)
mixed_saving ← 0
cur ← empty set
for i ← 1ton do
  add w[i] into cur
  if cur exceeds capacity then
    remove w[i] from cur
    mixed_saving = mixed_saving + saving(cur)
    cur ← empty set
    add w[i] into cur
  end if
end for
mixed_saving = mixed_saving + saving(cur)

return mixed_saving
```

2.3 Data collection/process

We collected our dataset from Google cluster data trace at Github, which include 29 days of data at May 2011 on a cluster of about 12.5k machines. There are 6 sets of .csv file in total and we mainly focused on task_events.csv which can be appropriately modeled into our problem. In the task_events table, each event has following useful features:

- Timestamp: in terms of microseconds.
- Job ID: used to identify each task.
- Event type: including one of the following type: (0) submit, (1) schedule, (2) evict, (3) fail, (4) finish, (5) kill, (6) lost, (7) update_pending, (8) update running. Here we treat (0) and (1) as the start of a task, (2), (3), (4), (5) and (6) as the stop of a task, and ignore (7) and (8).
- Machine ID: used to identify the location of each task.

We built vectors based on each machine. And each entry on the vector represents the number of running tasks during 2-hour duration. Thus, there are about total of 12500 vectors (equal to the number of machines) with length of 348(29*24/2). Note that the

data is not perfectly matched to the reality. Some task data are missing machine ID so it is useless to our problem, so we just ignore them.

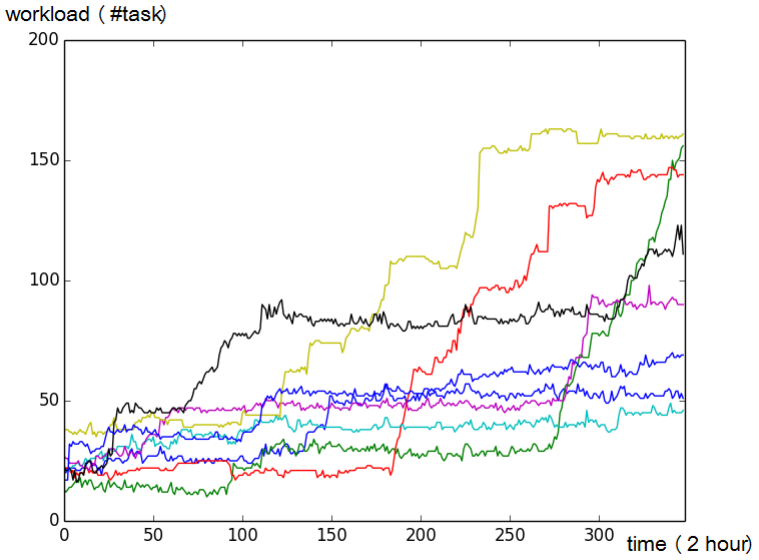

FIGURE 2.1: 8 vectors randomly selected from google cluster data

If increasing and decreasing of workloads happens at same timeslot between different vectors, we can reduce the total workload by mixing them. To show the potential of reducing the workloads, we have a figure of cumulative distribution function for vectors generated by google cluster data and showing how many pairs of increase/decrease overlaps as timeslot goes.

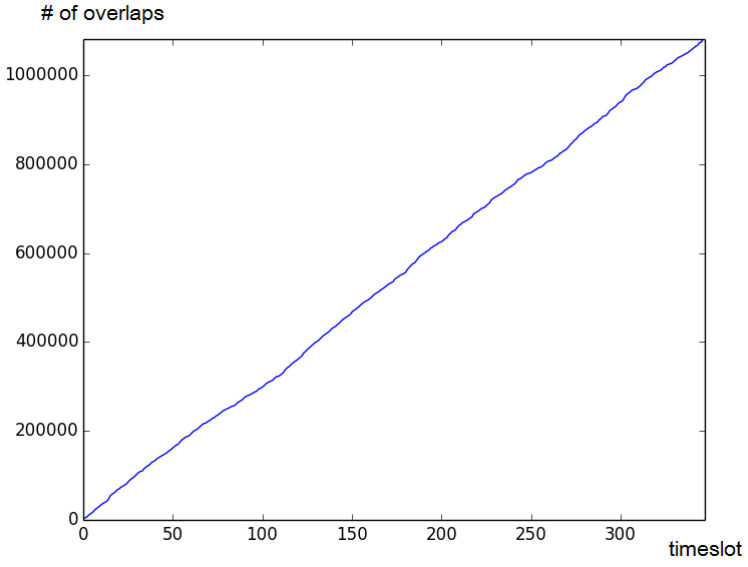

FIGURE 2.2: cumulative distribution function

From the figure 2.2, we can see there is significant potential to save workloads.

2.4 Experiment setup

We compare 3 methods by running them on different size of data. For a set of input data, we have following parameters which could influence the performance of the algorithm:

- number of vectors
- length of vectors
- size of capacity

To have a comparison, each time we set one of parameters as variable and fix the value of others. And we compare the performance of each algorithm by both time and accuracy.

2.5 Result

We firstly look at accuracy, then computational overhead:

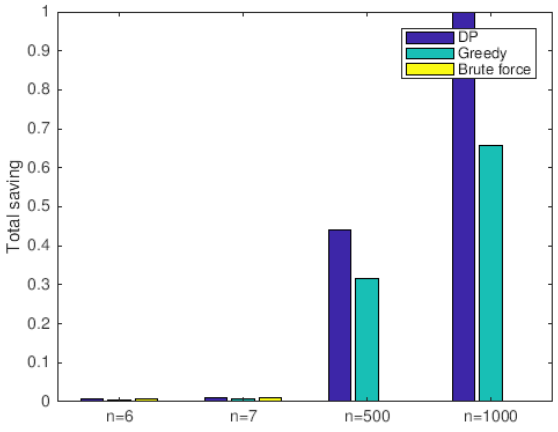

FIGURE 2.3: l=348, c=250, n varies (normalized)

From these figures, we can easily see that brute force only works on very small data. Dynamic programing can give us high accuracy result (which is equivalent to brute force when data is small), but its computational overhead is bad when n is large. Greedy, by contrast, has the shortest run time, and its result is worse than dynamic programing.

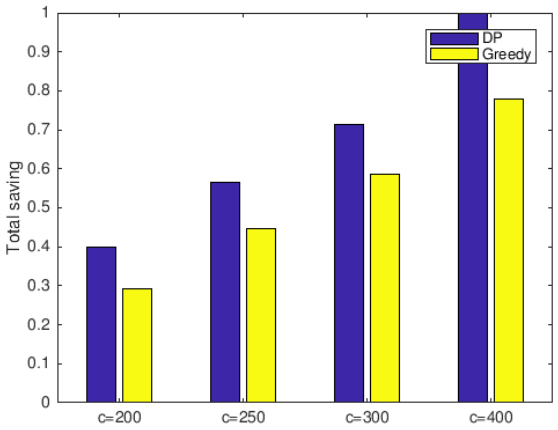

FIGURE 2.4: n=200, l=348, c varies (normalized)

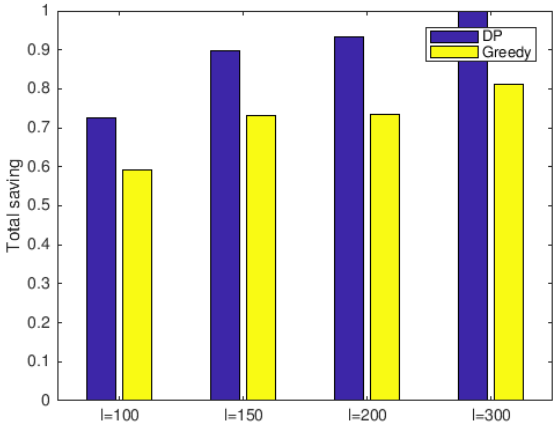

FIGURE 2.5: n=200, c=250, l varies (normalized)

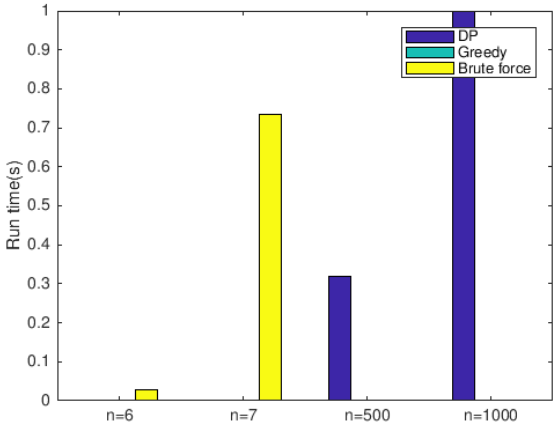

FIGURE 2.6: l=348, c=250, n varies (normalized)

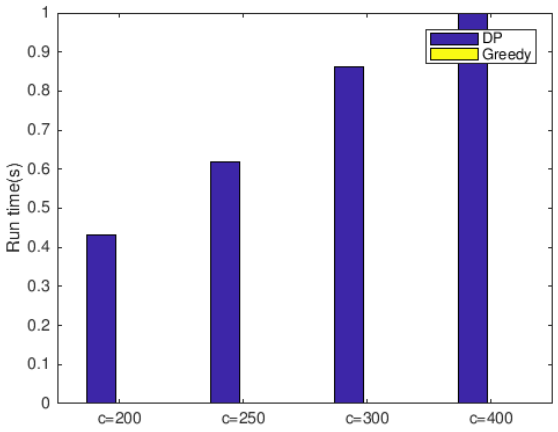

FIGURE 2.7: n=200, l=348, c varies (normalized)

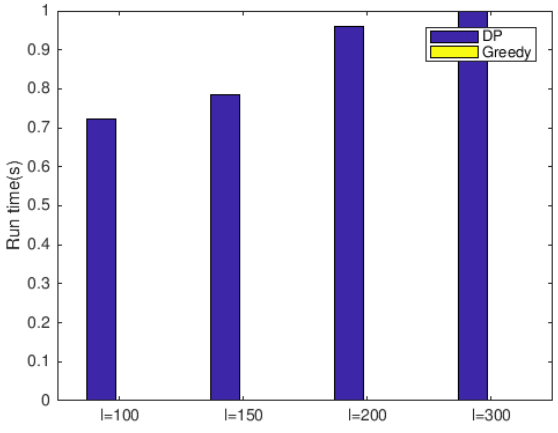

FIGURE 2.8: n=200, c=250, l varies (normalized)

Chapter 3

Heterogeneous services

3.1 Problem formulation

In the case of heterogeneous services, the type of each service are different. So we need more parameter to weigh workloads (i.e. boot time and transfer time). Furthermore, we cannot mix services to reduce reconfiguration cost. One alternative way is to transfer file/code between VMs. For example, if service A need one more VM to do some task and a VM on service B just finish its task and about to shut down, we can transfer files which are needed for the task from service A to service B, instead of boot a VM on service A and shut down the spare VM on service B. In general, the time spent by transferring data between VMs is less than booting up a VM. So the reconfiguration cost gets saved by this way.

One thing to mention is that in heterogeneous services, we don't need to care about capacity since we don't mix vectors anymore, nor do we add extra workload on each service, we only "recycle" spare VMs.

Notation	Explanation
T_i^{boot}	Time cost of booting up a VM in the ith service
$T_{i,j}^{transfer}$	Time cost of transferring file between VMs from the ith service to jth service ($i \neq j$), note that $T_{i,j}^{transfer}$ is not necessarily equal to $T_{j,i}^{transfer}$

Table 3.1: More notations

3.2 Algorithm description

Since we cannot mix services in this case, the method that uses the size of capacity to represent the status of the problem which we used in homogeneous case is not applicable anymore. Furthermore, as each vector is distinct in heterogeneous case, it requires exponential space to represent each status. Therefore, we abandon dynamic programming approach and find two greedy strategies instead.

3.2.1 Greedy: max saving first

At each time slot, for each service i which has growth trend(i.e. requires more VMs to do the task), we find a declined service j (i.e. requires less VMs) which gives us the most saving. In other word we find j which has the smallest $T_{i,j}^{transfer}$ for the service i. Then we transfer file from i to j as much as possible. If there is no more spare VM on j and i still need more VM, we find another service which has next most saving and transfer file from i to it. We repeatedly do this until i doesn't need any more VM or there is no service have spare VM. Then we move to next time slot and do the same or we finish all time slots.

3.2.2 Greedy: max boot time first

The basic idea is similar with the first strategy. Besides that, for each increasing vector i, instead of find j which gives us the most saving in current time slot, we find the j which has the largest T_j^{boot} and transfer file from i to j. The reason for us to adopt this strategy is that, if we need to boot VM on j in future, it would cost more time to do so, which is not we disired. So we keep then VMs on j running by transferring file to them to avoid booting them up in future.

We expect that this strategy performs better than the first one when most vectors have growth trends, as in such situation more VMs are forced to boot as time goes on.

3.3 Data collection/process

Dataset of vectors in homogeneous services can still be used here. However, in heterogeneous service we need more parameters: boot time of VM in each service and data transfer time between VMs in each pair of services. Since it is hard to create real cloud environment and get data from it, we build VMs on VM software and measure the data we need instead.

Algorithm 4 Greedy: max saving first

```
saving ← 0
for j ← 1tol − 1 do
  for i ← 1ton do
    if w[i][j + 1] ≤ w[i][j] then
      continue
    end if
    while True do
      max_saving ← −1
      max_index ← −1
      for k ← 1ton do
        if w[k][j + 1] ≥ w[k][j] then
          continue
        end if
        if T^boot[i] − T^transfer[i, k] > max_saving then
          max_saving ← T^boot_i − T^transfer[i, k]
          max_index ← k
        end if
      end for
      if max_index = −1 then
        break
      end if
      if w[i][j + 1] − w[i][j] ≤ w[max_index][j] − w[max_index][j + 1] then
        saving = saving + (T^boot[i] − T^transfer[i, max_index]) ∗ (w[i][j + 1] − w[i][j])
        w[max_index][j + 1] = w[max_index][j + 1] + w[i][j + 1] − w[i][j]
        w[i][j + 1] = w[i][j]
        break
      else
        saving = saving + (T^boot[i] − T^transfer[i, max_index]) ∗ (w[max_index][j] −
        w[max_index][j + 1])
        w[i][j + 1] = w[i][j + 1] − (w[max_index][j] − w[max_index][j + 1])
        w[max_index][j + 1] = w[max_index][j]
      end if
    end while
  end for
end for

return saving
```

Algorithm 5 Greedy: max boot time first

```
saving ← 0
for j ← 1tol − 1 do
  for i ← 1ton do
    if w[i][j + 1] ≤ w[i][j] then
      continue
    end if
    while True do
      max_boot_time ← −1
      max_index ← −1
      for k ← 1ton do
        if w[k][j + 1] ≥ w[k][j] then
          continue
        end if
        if T^boot[k] > max_boot_time then
          max_boot_time ← T^boot[k]
          max_index ← k
        end if
      end for
      if max_index = −1 then
        break
      end if
      if w[i][j + 1] − w[i][j] ≤ w[max_index][j] − w[max_index][j + 1] then
        saving = saving + (T^boot[i] − T^transfer[i, max_index]) ∗ (w[i][j + 1] − w[i][j])
        w[max_index][j + 1] = w[max_index][j + 1] + w[i][j + 1] − w[i][j]
        w[i][j + 1] = w[i][j]
        break
      else
        saving = saving + (T^boot[i] − T^transfer[i, max_index]) ∗ (w[max_index][j] −
        w[max_index][j + 1])
        w[i][j + 1] = w[i][j + 1] − (w[max_index][j] − w[max_index][j + 1])
        w[max_index][j + 1] = w[max_index][j]
      end if
    end while
  end for
end for

return saving
```

The VM software we use is VMware Workstation Pro 12.5.5. In order to measure the boot time of a VM, we create a VM with Ubuntu 64 bit operating system and 1GB memory. We want a stable result so measure 1000 times could be fair. Thus, we create a Linux shell and implement following approach:

1. Use command "systemd-analysis" to get boot time and write it into a text file.
2. Check the number of lines of the text file, if less than 1000, reboot.

We set the shell as a boot up item. So the VM will automatically reboot 1000 times and each time the boot time is recorded, which shows as figure 3.1.

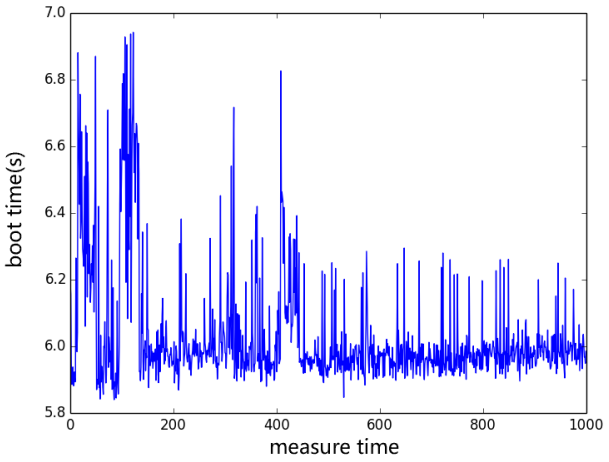

FIGURE 3.1: boot times

From the figure we can see the boot time is in the range 5.8s to 7.0s. The deviation which is under 1.2s is acceptable.

Regarding to data transfer time. We create another VM which has same configure as above and use SSH to transfer file between two VMs. According to official documents of several common web servers (IIS, Apache, nginx), the local disk usage of a web server is usually in range 10MB to 20MB (not including websites under it). Therefore, we use files in such range as test file and measure time cost of transfering them between VMs. The measurement result shows as figure 3.2.

As we expected, as the file becomes larger, the transfer time is increasing in the range of 1.2s to 2.6s. In addition, transfer time is in the same order as the boot time but obviously shorter than boot time, which is the desired result. Thus, we can generate inputs for the algorithm according to the measurement result we obtained above.

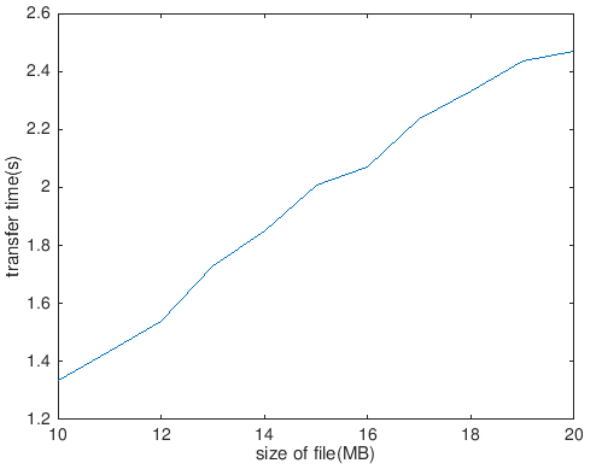

FIGURE 3.2: file transfer time

3.4 Experimental setup

As stated before, we don't care about capacity in heterogeneous case as we cannot mix vectors as a group. Besides of that, we do something similar with other two parameters (number and length of vectors) as homogeneous case: keep one fixed and another varies, and compare results by both accuracy and computational overhead. Since two strategies have the same time complexity, they are expected to have same run time as well.

Note that the boot time and file transfer time will not affect run time nor accuracy but final result. So we do not need to test with different value of them. According to the range of values we measured in section 3.3, we generate random numbers and use them as part of input.

3.5 Result

We firstly look at accuracy, then computational overhead:

From figure we can see, our algorithms work well for the input which is generated from real data. In comparison, the performance of two strategies are close. However, in the most part, the results from the first strategy are a little better. And run time of both are close as we expected.

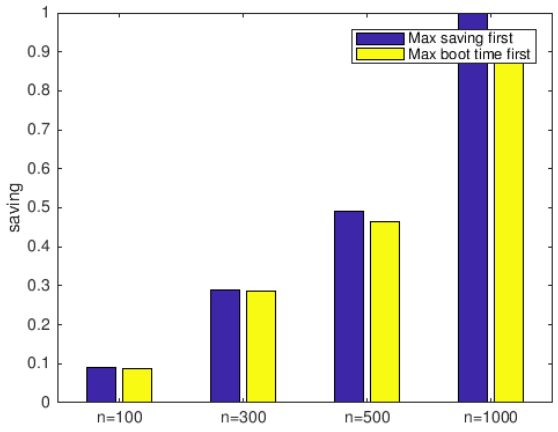

FIGURE 3.3: l=348, n varies (normalized)

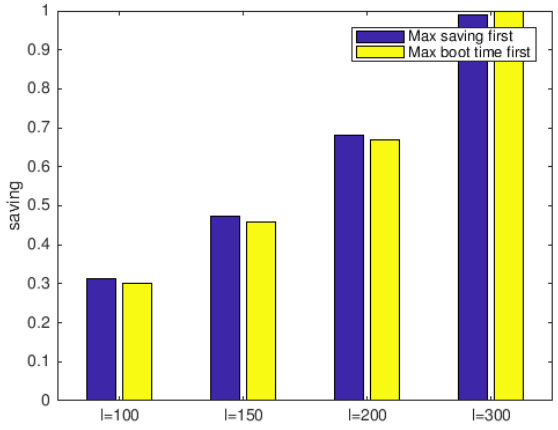

FIGURE 3.4: n=200, l varies (normalized)

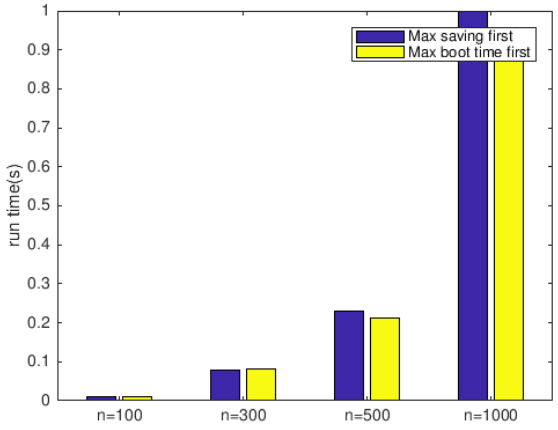

FIGURE 3.5: l=348, n varies (normalized)

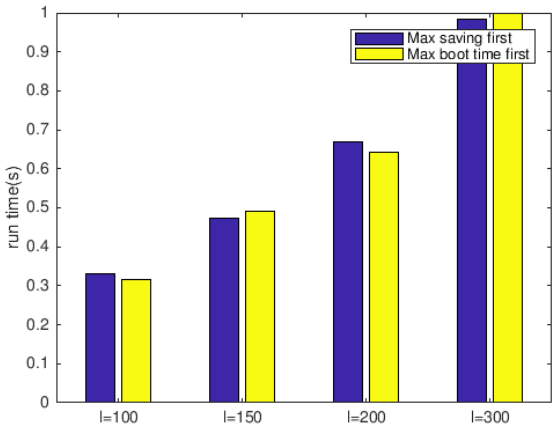

FIGURE 3.6: n=200, l varies (normalized)

Chapter 4

Conclusion

In this thesis, we described several algorithms to computing the decrements of reconfiguration cost in the cloud environment for both homogeneous and heterogeneous cases. For the homogeneous case, the main idea is mixing services. And for the heterogeneous case which services cannot be mixed, we transfer data between services instead.

In the homogeneous case, there is a trade off between time and accuracy (dynamic programming and greedy algorithm). We can choose the one which is more appropriate based on our actual demand.

We collect and generate data from the real world and use it as input to evaluate and justify our method. The result shows that our algorithms work correctly for the real world data.

Bibliography

[1] Compiling and installing apache http server version 2.4. https://httpd.apache.org/docs/2.4/install.html.

[2] Iis 7.0 and your hardware. https://msdn.microsoft.com/en-us/library/cc268240.aspx.

[3] Nginx documentation. http://nginx.org/en/docs/.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA, 1995.

[6] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[7] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[8] Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

[9] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[10] Milind Dawande, Jayant Kalagnanam, Pinar Keskinocak, F Sibel Salman, and R Ravi. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *Journal of combinatorial optimization*, 4(2):171–186, 2000.

[11] W Fernandez de La Vega and George S Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[12] Stefka Fidanova. Evolutionary algorithm for multiple knapsack problem. In *IN PROCEEDINGS OF PPSN-VII, SEVENTH INTERNATIONAL CONFERENCE ON PARALLEL PROBLEM SOLVING FROM NATURE, LECTURE*. Citeseer, 2002.

[13] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[14] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[15] Gueyoung Jung, Tridib Mukherjee, Shruti Kunde, Hyunjoo Kim, Naveen Sharma, and Frank Goetz. Cloudadvisor: A recommendation-as-a-service platform for cloud configuration and pricing. In *Services (SERVICES), 203 IEEE Ninth World Congress on*, pages 456–463. IEEE, 2013.

[16] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 188–193. ACM, 1994.

[17] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

[18] Silvano Martello and Paolo Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27(2):93–112, 1981.

[19] Silvano Martello and Paolo Toth. 0/1 multiple knapsack problem. In *Knapsack problems: algorithms and computer implementations*, chapter 6, pages 157–187. 1990.

[20] Pinal Salot. A survey of various scheduling algorithm in cloud computing environment. *International Journal of Research in Engineering and Technology*, 2(2):131–135, 2013.

[21] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96. ACM, 2009.

[22] Yi Wei and M Brian Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72–75, 2010.

[23] Baomin Xu, Chunyan Zhao, Enzhao Hu, and Bin Hu. Job scheduling algorithm based on berger model in cloud environment. *Advances in Engineering Software*, 42(7):419–425, 2011.