Performance Comparison for Resource Allocation Schemes using Cost Information in Cloud

Takahiro KOITA [1] and Kosuke OHARA [1]
[1] Doshisha University Graduate School of Science and Engineering,
Tataramiyakotani 2-6-2, Kyotanabe, Kyoto, 610-0321, Japan

Abstract. A wide variety of different types of virtual computer are available in cloud computing, each with different usage costs for processing performance and time. Consequently, similar processing tasks can incur different processing times and processing costs depending on the choice of method used to accomplish them. Since the amounts of time and money that can be spent on processing are not infinite, the processing time and usage costs must be reduced as much as possible. In this study, we investigate the allocation of resources in a cloud computing environment with the aim of achieving lower processing times and usage costs.

Keywords: cloud computing, Amazon EC2, resource allocation

1. Introduction

In recent years, the spread of cloud computing has made it easy for users to work with very substantial computer resources. One cloud computing environment is Amazon EC2 (Amazon Elastic Compute Cloud) [1], which is implemented as a so-called IaaS (Infrastructure as a Service). In this environment, it is possible to use computer resources via the Internet in the form of virtual computers called “instances”. Instances can be hired in hourly units, and are billed according to how many hours they are used for. There are several types of instance, which differ in terms of their hourly usage costs according to parameters such as CPU performance and memory capacity. Consequently, the exact same process can incur different processing times and usage costs depending on the choice of instance. For example, consider the following situation. In Amazon EC2, the CPU processing performance is expressed in units called ECUs (EC2 Compute Units). Suppose an instance A has a CPU performance of 1 ECU and a usage cost of $0.10/day, and another instance B has a CPU performance of 5 ECU and a usage cost of $0.30/day. When these two instances are used to perform a process that takes 10 hours on a CPU with a performance of 1 ECU, instance A will take 10 hours to complete the process at a cost of $1.00, while instance B will take 2 hours at a cost of $0.60. Thus the processing times and usage costs can differ depending on how instances are used, even for the same process. Since the amounts of time and money that can be spent on processing are not infinite, it is desirable to reduce the processing time and usage costs as much as possible. To reduce the processing times and usage costs, the processing time and usage costs must be estimated in order to predict what sort of processing time and usage cost will be incurred before allocating computer resources to a process. By allocating resources based on this prediction, the processing time and usage costs can be reduced, so it is important to investigate methods for allocating resources in cloud computing environments.

Methods for allocating resources based on cost have been studied in grid computing before the advent of cloud computing [2][3]. In grid computing, various resource allocation methods have been investigated, including splitting and distributing loads across groups of computer resources [4]. However, in grid computing, in addition to the need to consider the costs incurred by users and suppliers of computer resources, the situation is also complicated by the fact that each provider offers resources under different conditions. This differs from cloud computing environments where computer resources are provided under fixed conditions by a single company. In cloud computing, researchers are studying how to add and delete computer resources according to load, and how to use cloud computing environments from a network perspective [5][6]. However, most of the research into methods for allocating resources in cloud computing have centered around load distribution, and few consider the cost aspects of resource allocation.

In this study, we evaluate the variation of cost performance and processing time in the Amazon EC2 could computing environment, and we investigate a resource allocation method that takes costs into consideration.

2. Amazon EC2

2.1. Overview

Amazon EC2 is a cloud computing environment where virtual computers are made available via the Internet. These virtual computers are called “instances”. The data centers that actually contain the computer resources that provide an instance are represented as regions, and the regions can be selected by users.

An instance includes hardware resources (i.e., computer resources in a data center) and software resources, and users are free to configure their own environments inside each instance. Each instance has a different performance and hourly usage cost, and users are able to select instances according to their needs. Details of each instance are shown in Table 1. Here, the usage fees are quoted for instances where the region is North California and the OS is Windows Server. The CPU performance of each instance is expressed as an ECU value, where 1 ECU corresponds to the CPU performance of a single 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. The ECU value is basically fixed for each instance, but a micro instance can increase its CPU performance to a maximum of 2 ECU when heavily loaded. The available OS platforms are 32-bit systems for m1.small and c1.medium, 64-bit systems for m1.large, m1.xlarge, m2.xlarge, m2.2xlarge, m2.4xlarge and c1.xlarge, and either 32-bit or 64-bit systems for t1.micro. The prefixes m1, m2 and so on represent the instance type, and are described by Amazon as follows:

- m1 (standard instance): Instances in this family are suitable for almost all applications.
- t1 (micro instance): Instances in this family provide a small amount of consistent CPU resources and can provide increased CPU capacity in short bursts when additional cycles are available. They are suitable for low-throughput applications and websites that consume a certain amount of compute cycles periodically.
- m2 (high-memory instance): Instances in this family provide a large amount of memory for high throughput applications, such as database and memory caching applications.
- c1 (high-CPU instance): Instances in this family have proportionally more CPU resources than memory (RAM), and are suitable for applications that require large amounts of computation.

Table 1: Details of each instance

Instance	ECU	No. of cores	ECUs per core	Memory (GB)	I/O performance	Storage (GB)	Hourly usage fee ($/h)
m1.small	1	1	1	1.7	Standard	160	0.13
m1.large	4	2	2	7.5	Fast	850	0.52
m1.xlarge	8	4	2	15	Fast	1690	1.04
t1.micro	Up to 2	1	Up to 2	0.613	Fast	Pay-as-you-go	0.035
m2.xlarge	6.5	2	3.25	17.1	Standard	420	0.69
m2.2xlarge	13	4	3.25	34.2	Fast	850	1.38
m2.4xlarge	26	8	3.25	68.4	Fast	1690	2.76
c1.medium	5	2	2.5	1.7	Standard	320	0.31
c1.xlarge	20	8	2.5	7	Fast	1690	1.24

2.2. Problems in the use of Amazon EC2

In Amazon EC2, users are free to choose from a variety of different instances. However, in selecting an instance, decisions have to be based only on the instance's performance value and its hourly usage cost. Compared with m1.small, a c1.medium instance has an hourly usage cost that is three times larger, but offers five times as much CPU performance. Thus, when performing processes that depend on CPU performance alone, the processing time of m1.small would be five times that of c1.medium. However, since the usage costs differ by a factor of 2.4, the processing time and usage cost are both smaller for c1.medium. For a process that depends only on CPU performance, we can predict which instance will have the lowest processing time and usage cost simply from the performance value and hourly usage cost as described above. However, real processes do not necessarily depend on CPU performance alone, and even those that do are affected by other factors such as memory and I/O performance. It is therefore difficult to judge which instance should be selected for a particular process solely on the basis of the performance value and hourly usage cost, and a poor choice of instance can lead to unnecessarily large processing times and usage costs.

3. Evaluation experiment

To reduce the processing time and usage cost, it is necessary to investigate a resource allocation method to choose which instance to allocate to a process. Resource allocation methods must be investigated based on evaluated values in each instance, such as cost performance and processing times. Consequently, in the evaluation experiment, we evaluated the cost performance and processing time of each instance. To evaluate the cost performance and processing time, we ran a prime number search process, a matrix arithmetic process and a file loading process on each instance and evaluated the cost performance and processing time in each case. The prime number search process, matrix arithmetic process and file loading process were evaluated by performing each of them 100 times, so the cost performance is defined as the number of processes that can be executed for 1 dollar. Each process was run 100 times in a row, and the processing time was the actual time taken between the first time the process is run and the end of the 100th time it is run. We chose North California for the Amazon EC2 region, and we used Windows Server 2008 for the OS. Although Amazon EC2 is billed in hourly units, we

calculated a per-second usage cost, even for periods of less than one hour, in order to perform a detailed evaluation of each instance.

3.1. Prime number search process

To evaluate each instance in terms of CPU performance, we evaluated the cost performance and processing time using a process to search for the prime numbers below 500,000. The number of CU cores per instance ranged from 1 to 8. To use all the CPU cores in each instance, the prime number search process was run simultaneously in 10 threads. The cost performance and processing time were obtained from the results of repeating this prime number search process 100 times. Figures 1 and 2 show the cost performance and processing time of each instance.

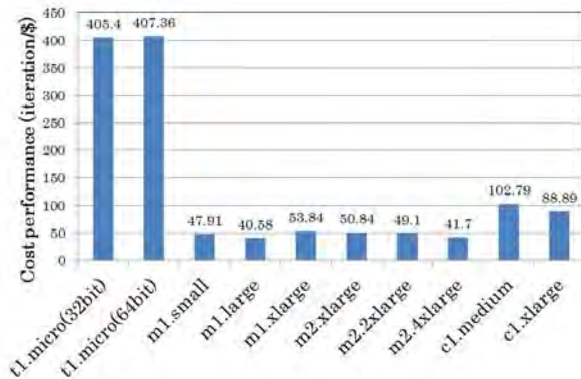

Figure 1: Cost performance of prime number search

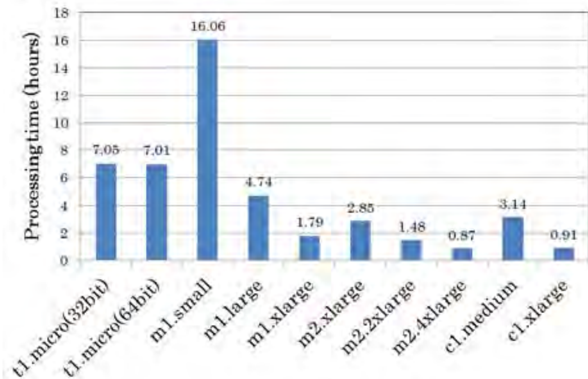

Figure 2: Processing time of prime number search

The cost performance of the prime number search process was very large for the t1.micro (32 bit) and t1.micro (64 bit) instances, followed by c1.medium and c1.xlarge. The smallest processing time was achieved with the m2.4xlarge instance, followed by c1.xlarge and m2.2xlarge.

Table 2 shows the average, maximum, minimum and standard deviation of the time taken by each prime number search process. Apart from t1.micro (32 bit) and t1.micro (64 bit), the average processing times were more or less the same in each instance. In the instances apart from t1.micro (32 bit) and t1.micro (64 bit), the processing time remains more or less constant from the first time the prime number search process is run, but in t1.micro (32 bit) and t1.micro (64 bit), the processing time increased up to almost the 30th iteration, and then became more or less constant.

Table 2: Processing time of each instance in the prime number search process

Instance	Average (s)	Maximum (s)	Minimum (s)	Standard deviation (s)
m1.small	578.06	581	577	1.057
m1.large	170.61	172	170	0.508
m1.xlarge	64.29	66	63	0.516
t1.micro (32 bit)	253.72	265	208	10.896
t1.micro (64 bit)	252.50	264	208	10.667
m2.xlarge	102.63	103	102	0.483
m2.2xlarge	53.13	54	52	0.416
m2.4xlarge	31.28	32	31	0.449
c1.medium	112.98	114	112	0.374
c1.xlarge	32.66	33	32	0.474

3.2. Matrix arithmetic process

To evaluate each instance in terms of memory performance, we evaluated the cost performance and processing time using a process involving a 16000×16000 matrix arithmetic operation. To use up the memory in each instance, the matrix arithmetic process was performed by dynamically reserving memory areas. The cost performance and processing time were obtained from the results of repeating this matrix arithmetic process 100 times. Figures 3 and 4 show the cost performance and processing time of each instance.

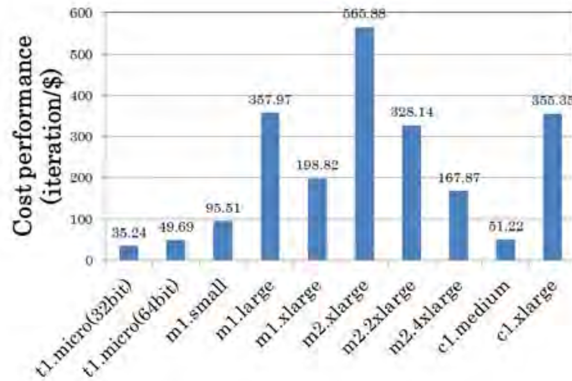

Figure 3: Cost performance of matrix arthmetic

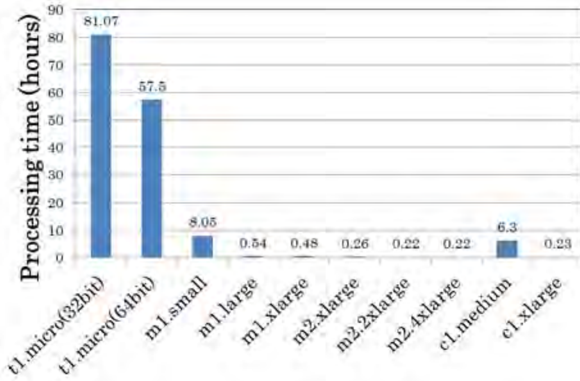

Figure 4: Processing time of matrix arithmetic

The cost performance of the matrix arithmetic process was largest for the m2.xlarge instance, followed by the m1.large, c1.xlarge and m2.2xlarge instances. The processing time was smallest for the m2.2xlarge and m2.4xlarge instances, followed by the c1.xlarge and m2.xlarge instances.

Table 3 shows the average, maximum, minimum and standard deviation of the time taken by each matrix arithmetic process. For t1.micro (32 bit) and t1.micro (64 bit), we observed a phenomenon whereby the processing time for each matrix arithmetic process suddenly increased. Also, this phenomenon occurred at different times and after different numbers of iterations in t1.micro (32 bit) and t1.micro (64 bit).

A small amount of variation was also observed for m1.small and c1.medium, but there was no sudden increase in processing time and instead the processing times went up and down in the vicinity of the average value.

Table 3: Processing time of each instance in the matrix arithmetic process

Instance	Average (s)	Maximum (s)	Minimum (s)	Standard deviation (s)
m1.small	289.94	400	577	22.195
m1.large	19.34	21	16	0.919
m1.xlarge	17.41	19	16	0.991
t1.micro (32 bit)	2918.37	34759	788	6559.240
t1.micro (64 bit)	2070.08	22452	832	2524.616
m2.xlarge	9.22	13	7	1.293
m2.2xlarge	7.95	9	7	0.357
m2.4xlarge	7.77	8	7	0.421
c1.medium	226.70	308	194	20.329
c1.xlarge	8.17	9	8	0.376

4. Discussion

In the prime number search process, the cost performance was very large for the t1.micro (32 bit) and t1.micro (64 bit) instances, but this is due to the low hourly rate for the t1.micro instances. The hourly rate for a t1.micro instance is $0.035, which is 3.7 times smaller than the next cheapest instance (m1.small: $0.13), and 78.9 times smaller than the most expensive (m2.4xlarge: $2.76). Thus, the cost performance of a t1.micro instance is greater as long as its processing times are less than 3.7 times or 78.9 times are large as these other instances. To allow a comparison to be made between all the instances and not just t1.micro, when comparing instances that have different hourly usage costs, if the quotient of the processing time does not exceed the quotient of the hourly usage cost then the cost performance is higher for the instance with the lower hourly usage cost. Although there was no large difference in cost performance between m1.small and m2.2xlarge in the prime number search process, there was a large difference in processing times. Even if there is no difference in cost performance, there can sometimes be a large difference in processing time.

With regard to the processing times in the matrix arithmetic process, there was hardly any difference between the m2.2xlarge, m2.4xlarge and c1.xlarge instances. When there is no difference in processing time, differences can arise in cost performance due to differences in the hourly usage cost, so in a resource allocation method that takes cost into consideration, an instance with a low hourly usage cost should be selected.

In this section, we allocates instances to a virtual process based on the cost performance figures obtained in the evaluation experiment, and compared the resource allocation methods by estimating the processing time and usage cost. However, since the processing time can suddenly change in t1.micro instances, it is difficult to use resource allocation methods that examine the cost performance and processing time. We therefore excluded t1.micro instances from the scope of comparison in the investigation of resource allocation methods described

below. To simulate a comprehensive process, we performed a process comprising 1011 iterations of the prime number search process, 4507 iterations of the matrix arithmetic process, and 2208 iterations of the file loading process. These numbers of iterations were obtained by determining the average value of the cost performance of instances other than t1.micro in each process, and calculating the number of iterations for which the contribution of each process becomes equal based on this average value. We also decided to run the prime number search, matrix arithmetic and file loading processes simultaneously. The calculated results are shown in Table 5.

Table 5: Estimated results of resource allocation methods

API name	Usage cost ($)	Processing time (hours)
m1.small	48	369.23
m1.large	25	48.08
m1.xlarge	24	23.08
m2.xlarge	20	28.99
m2.2xlarge	21	15.22
m2.4xlarge	42	15.22
c1.medium	88	283.87
c1.xlarge	35	28.23

From these results, the instances can be classified into the following four groups:

- Low usage costs and short processing times:
  m1.xlarge, m2.xlarge, m2.2xlarge
- Low usage costs, long short processing times:
  m1.large
- High usage costs, short processing times:
  m2.4xlarge, c1.xlarge
- High usage costs and long processing times:
  m1.small, c1.medium

Therefore, in processes with average requirements for CPU performance, memory performance and I/O performance, it is predicted that the usage costs and processing times can be reduced by using a m1.xlarge, m2.xlarge or m2.2xlarge instance.

5. Summary and future works

In this study, we have discussed a cost-based resource allocation method for Amazon EC2. Since it is difficult to determine which instance should be used in situations where the performance values and hourly usage rates are not published, we evaluated the cost performance and processing time in each instance and performed trial calculations of a resource allocation method based on the results. As an issue for further study, it will be necessary to investigate the results of these trial calculations in real environments. Also, since it is possible that this experiment did not make full use of the computer resources in instances with high processing performance, it will be necessary to perform an evaluation experiment with higher processing loads and to combine the results with those of the present study to investigate cost-based resource allocation methods.

6. References

[1] Amazon Elastic Compute Cloud (Amazon EC2), Amazon web services, 2008.

[2] Rajkumar Buyya, Economic-based Distributed Resource Management and Scheduling for Grid Computing, PhD Thesis, Monash University, 2002.

[3] David Abramson, Rajkumar Buyya, and Jonathan Giddy, A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker, Future Generation Computer Systems, Volume 18, Issue 8, pp. 1061–1074, Elsevier Science, 2002.

[4] Minyi Guo and Baoliu Ye, Automatic parallel distribution and optimization of applications in a grid computing environment, Telecommunications Advancement Foundation, research report No. 21, pp. 475–484, 2006.

[5] Ryota Miyagi, Minoru Ikebe, Atsuo Inomata, Kazutoshi Fujikawa and Hideki Sunahara, Proposal and evaluation of a dynamic resource allocation system that adapts to server load in cloud environments, Transactions of the IPSJ, Vol. 2011-IOT-12, No. 4, 2011.

[6] Taiki Morikawa, Atsuo Inomata, Minoru Ikebe, Yoshihiro Okamoto, Satoshi Uda, Kazutoshi Fujikawa and Hideki Sunahara, A mechanism for dynamic resource allocation based on Web server load in cloud computing environments, Transactions of the IPSJ, Vol. 2010-IOT-8, No. 20, 2010.