Comparing AWS deployments using model-based predictions ⋆

Einar Broch Johnsen, Jia-Chun Lin, Ingrid Chieh Yu

Department of Informatics, University of Oslo, Norway
{einarj,kellylin,ingridcy}@ifi.uio.no

Abstract. Cloud computing provides on-demand resource provisioning for scalable applications with a pay-as-you-go pricing model. However, the cost-efficient use of virtual resources requires the application to exploit the available resources efficiently. Will an application perform equally well on fewer or cheaper resources? Will the application successfully finish on these resources? We have previously proposed a model-centric approach, ABS-YARN, for prototyping deployment decisions to answer such questions during the design of an application. In this paper, we make model-centric predictions for applications on Amazon Web Services (AWS), which is a prominent platform for cloud deployment. To demonstrate how ABS-YARN can help users make deployment decisions with a high cost-performance ratio on AWS, we design several workload scenarios based on MapReduce benchmarks and execute these scenarios on ABS-YARN by considering different AWS resource purchasing options.

1 Introduction

Cloud computing is currently the main driver of an on-going change in how companies develop and exploit software by providing utility computing services [4]: IT resources and applications over the Internet are delivered on demand with a pay-as-you-go cost model. Cloud computing enables the infrastructure on which a software is deployed to be specialized to the needs of the software and even adapted at runtime by means of various adjustments of the configuration of the infrastructure. However, if the software does not adapt in an agile way to the changing deployment decisions (which are decisions of determining the capacity of computation resources, the scale of computation resources, or other parameters), we may experience a wasteful over-provisioning of resources or require substantial reengineering of the software, which might increase either the operational or the development costs of the software or both. Shifting deployment decisions from the deployment phase to the design phase of a software development process can significantly reduce such costs by performing model-based validation of the chosen decisions during the software design [10]. This shift

⋆ Supported by the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems* (http://www.hyvar-project.eu)

requires that deployment decisions can be captured in formal models of the virtualized and distributed software and that these models enable us to assess the impact of different deployment decisions on the performance and operational cost of our software.

There are many providers of cloud computing technologies on the market today to offer a plethora of services in cloud ecosystems for both computing and data storage, e.g., virtual machine instances, containers, ready-made solutions for resource management and auto-scaling, service endpoints, as well as support for data storage, databases, and caching at many different costs. It is not easy to select a solution which best balances performance and incurred costs for a particular application. Even for very basic virtual machine instances, a company such as Amazon offers approximately 40 different so called instance types with different prices and resource specifications for on-demand virtual servers alone.

In this paper, we compare—at the modeling level—the effect of selecting different instance types for virtual servers from Amazon Web Services (AWS) in terms of performance and accumulated cost. Using a model-based approach, we design a set of workload scenarios consisting of a number of MapReduce benchmarks and consider different AWS instance type deployments to execute these scenarios so as to study the impacts of these deployments in terms of performance, cost, workload completion, etc.

The method used in this paper for model-based comparisons is based on the authors' previous work on ABS-YARN [15]. ABS-YARN is a highly configurable modeling framework for applications running on Apache YARN [21], a popular open-source distributed software framework for big data processing on cloud environments provided by vendors such as Amazon, HP, IBM, Microsoft, and Rackspace. ABS-YARN is defined in Real-Time ABS [5], a formal executable language for modeling deployed virtualized software by introducing a separation of concerns between the resource costs of the execution and the resource provisioning at (virtual) locations [14]. The focus of ABS-YARN is on obtaining results based on easy-to-use rapid prototyping, using the executable semantics of Real-Time ABS, defined in Maude [7], as a simulation tool for ABS-YARN. In previous work [15], we have shown through comprehensive experiments that ABS-YARN provides a satisfactory modeling accuracy as compared with a real YARN cluster. In this paper, we will base on the designed scenarios to demonstrate how users can utilize ABS-YARN to better understand the cost-performance tradeoffs between different AWS resource purchasing options and make appropriate purchase decisions.

Paper overview. Sect. 2 discusses AWS, focusing on how to understand the specification of instance types for on-demand virtual servers. Sect. 3 describes YARN and ABS-YARN. Sect. 4 presents the modeling and comparison of different AWS instances in several scenarios. Sect. 5 discusses related work and Sect. 6 concludes the paper.

Table 1. The specifications of five AWS instance types.

Instance type	vCPU	ECU	Memory (GB)	Price per hour
m4.large	2	6.5	8	$ 0.143
m4.xlarge	4	13.0	16	$ 0.285
m4.2xlarge	8	26.0	32	$ 0.57
m4.4xlarge	16	53.5	64	$ 1.14
m4.10xlarge	40	124.5	160	$ 2.85

2 Amazon Web Services

Amazon Web Services (AWS) is the dominating player on today's market for cloud computing with an approximate 30% market share for Q4, 2015 [6], especially for Infrastructure as a Service (IaaS). AWS offers an extensive ecosystem of services to help users to deploy, scale, and manage virtualized services on AWS infrastructure, including computing resources such as virtual servers, containers, load balancers, auto-scalers, etc. However, the pricing options in this ecosystem are complex: considering only the instance types of AWS itself (i.e., the different kinds of virtual machines), different types of instances may be acquired on demand, with no, partial, or all upfront payments with commitment for shorter or longer time periods, or at spot price[1].

As a customer of cloud computing services, it is important to understand the consequences that specific choices in this ecosystem may have for your software, both with respect to the *performance* of the software and to its incurred *operational costs*. In this paper, we focus on *on-demand* instance types, which do not require long-term commitments and therefore might be a good option for most customers who only run their applications sporadically.

Within this range of on-demand instance types, there are instance types which are suggested for general purpose applications as well as for compute-intensive and memory-optimized applications. Currently, there are around 40 different instance types available for on-demand instances. However, it is non-trivial to answer questions such as "what kind of instances should we choose for an application to achieve the best trade-off between performance and cost?".

Table 1 lists five on-demand instance types which all belong to AWS m4 category. M4 instances are the latest generation of general purpose instances, which provides a balance of compute, memory, and network resources, and it is a good choice for many applications [2]. Each row of Table 1 presents the specification of a particular instance type. We can see that each instance type roughly doubles the resources of the instance type on the previous row, but the instance type "m4.10xlarge" deviates slightly from this format. It is interesting to observe that deploying software on a utility computing model is different from how developers are used to think about CPU resources. Using the cloud, on-demand processing capacity is rented by the hour whereas, using a traditional provisioning model, a specific processor is bought with a long lifespan. AWS uses

[1] For details, see `https://aws.amazon.com/ec2/pricing`.

commodity hardware which is continually being replaced, so different hardware may be used to deliver the same virtual machine instances [1]: “Our goal is to provide a consistent amount of CPU capacity no matter what the actual underlying hardware.” The actual processing capacity of an instance type is thus given in terms of *elastic compute units* (ECU) whereas the column vCPU suggests a traditional understanding in terms of hardware.

3 YARN and ABS-YARN

In this section, we will briefly introduce YARN and then describe how ABS-YARN models YARN.

YARN [21] (short for Yet Another Resource Negotiator) is an open-source software framework supported by Apache for distributed processing and storage of high data volumes. It inherits the advantages of its well-known predecessor Hadoop [3], including resource allocation, distributed data processing, fault tolerance, and data replication. YARN further improves Hadoop’s limitations in terms of scalability, multi-tenancy support, cluster utilization, and reliability.

YARN supports the execution of different types of applications (or jobs), including MapReduce, graph, and streaming. Each job is divided into a set of smaller tasks which are executed in parallel on a cluster of machines. The key components of a YARN cluster are as follows:

- *ResourceManager* (RM): RM allocates resources to various competing jobs and applications in a cluster. The scheduling provided by RM is job level, rather than task level. Thus, RM does not monitor each task’s progress or restart any failed task.
- *ApplicationMaster* (AM): An AM is an instance of a framework-specific library class for a particular job. It acts as the head of the job to manage the job’s lifecycle, including requesting resources from RM, scheduling the execution of all tasks of the job, monitoring task execution, and re-executing failed tasks.
- *Slave nodes*: Each slave node provides both computation resources and storage to execute tasks and store data.
- *Containers*: Each container is a logical resource collection of a particular slave node (e.g., 1 CPU and 2GB of RAM). Clients can specify the resource requirement of a container when they submit jobs to RM and run any kind of application on containers.

Given RM and a set of slave nodes, a YARN cluster provides both computation resources and storage capacity to execute applications and store data. Assume that RM only has one queue, the execution of a job on YARN is as follows:

1. Whenever RM receives a job request from a client, RM follows a FIFO scheduling policy to find a container from an available slave and initiate the AM of the job on the container.

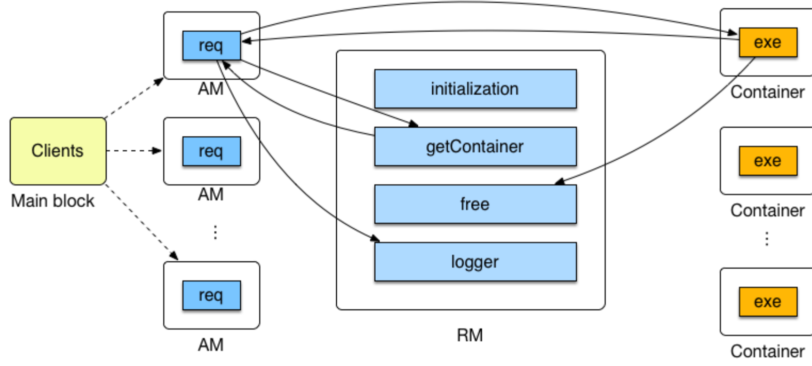


Fig. 1. The structure of the ABS-YARN framework.

2. Once the AM has been initiated, it starts requesting containers from RM based on the container resource requirements and the number of tasks of the job. Each task will be run on one container.
3. When RM receives a container request from the AM, it inserts the request into its queue and follows its job scheduling algorithm to allocate the desired container from an available slave node to the AM.
4. Upon receiving a container, the AM executes one task of the job on the container and monitors this task execution. If a task fails due to some errors such as an underlying container failure or slave node failure, the AM will re-request a container from RM to restart the task.
5. When all tasks of a job finish successfully, implying that the job is complete, the AM notifies the client about the completion.

ABS-YARN [15] is an executable model of YARN written in Real-Time ABS [5], which is a formal, executable, object-oriented language for modeling distributed systems by means of concurrent object groups [13]. Real-Time ABS uses *deployment components* to capture a location with a given resource specification in the deployment architecture, on which a number of concurrent objects can be deployed [14]. ABS-YARN follows the same execution flow as a YARN cluster.

Figure 1 shows the ABS-YARN architecture, which consists of interfaces AM, RM, and Container to model AM, RM, and containers, respectively. Interface RM has four methods. When a user starts ABS-YARN, method initialization initializes the entire cluster environment, including RM and slaves. Each slave has its own CPU, speed, and memory capacities. After the initialization, the cluster can start serving client requests. The method getContainer allows an AM to obtain containers with given resource requirements from RM. The method free is used to release container resources, and the method logger is used to record job execution statistics, including job ID and job execution time.

Currently, ABS-YARN focuses on the modeling of MapReduce jobs, which are the most common jobs in YARN. Each MapReduce job has two phases to process data: map phase and reduce phase. In the map phase, all map tasks are executed in parallel. When all the map tasks have completed, the reduce tasks are executed. The job is completed when all the map and reduce tasks have finished. Interface AM has only one method req, which is designed to request containers from RM and then ask the allocated containers to the execute the tasks of its job. For an AM, the total number of times that method req is called corresponds to the total number of the map tasks and reduce tasks of a job (e.g., if a job is divided into 10 map tasks and one reduce task, this method will be called 11 times). When all map tasks of the job have successfully completed, the AM proceeds with a container request to run the reduce task of the job. Only when all map and reduce tasks have completed successfully, the job is considered completed.

Interface Container has method exe to execute a task. The execution time of a task in a real YARN cluster might be influenced by many factors, e.g., the size of the processed data and the computational complexity of the task. To reduce the complexity of modeling the task execution time, ABS-YARN adopts the cost annotation functionality of Real-Time ABS to associate cost to the execution of a task. Hence, the task execution time will be the task cost divided by the CPU capacity of the container that executes the task.

ABS-YARN allows users to freely determine the scale and resource capacity of a YARN cluster by configuring the following parameters:

- the number of slave nodes in the cluster,
- the CPU capacity of each slave node, and
- the memory capacity of each slave node.

In addition, to support dynamic and realistic modeling of job execution, ABS-YARN also allows users to define the following parameters:

- Number of clients submitting jobs
- Number of jobs submitted by each client
- Number of map tasks and reduce tasks per job
- Cost annotation for each task
- CPU and memory requirements for each container
- Job inter-arrival pattern. Users can determine any kind of job inter-arrival distributions in ABS-YARN.

In the next section, we will apply ABS-YARN to study different instance types provided by AWS.

4 AWS Instances Study

As discussed in Sect. 2, AWS provides different kinds of instances to meet the resource requirements of different customers, including the on-demand instances,

reserved instances, and spot instances. Here, we focus on comparing the on-demand instances which we believe are the best suited for sporadic YARN jobs because renting this type of instance does not need a long-term commitment. Different operating systems, such as Linux, Red Hat, and Windows, are also available for the on-demand instances. In this paper, we focus on Linux because this operating system is supported by YARN and commonly used by the software development community. We chose EU (Frankfurt) as the region of instance and consider five general-purpose instance types. Table 1 in Sect. 2 lists the corresponding specifications and prices.

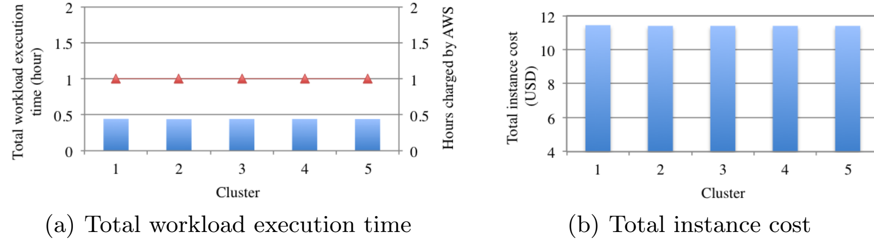
Table 2. The average map-task execution time (AMT) and average reduce-task execution time (ART) of each benchmark job [15].

Benchmark	AMT (sec)	ART (sec)
WordCount	295.47	430.24
WordMean	139.98	201.11
WordSD	238.46	312.38
GrepSort	37.38	62.06
GrepSearch	173.92	205.94

To compare the five chosen AWS instance types, we create realistic workloads using the following MapReduce benchmark jobs as in [15]: WordCount, WordMean, WordSD, GrepSort, and GrepSearch. Each benchmark job processes 1 GB of enwiki data [8] with 128 MB block size (which is the default block size of YARN [16]). Therefore, each job has 8 (=1GB/128MB) map tasks and one reduce task, implying that 9 containers are required to execute each job. Based on the average task execution time of each benchmark job shown in Table 2, we configure the ABS-YARN framework with the corresponding task cost annotation for each benchmark job (following [15]). We use 350 seconds as a threshold to classify the five benchmark jobs based on their execution time as below:

- Type 1: The summation of the AMT and ART of a job is larger than or equal to the threshold.
- Type 2: Otherwise.

Hence, the WordCount, WordSD, and GrepSearch jobs are classified as type 1 (i.e., long jobs), whereas the WordMean and GrepSort jobs are classified as type 2 (i.e., short jobs). Based on this information, we create two scenarios to compare the five AWS instance types. In the first scenario, we create a non-urgent workload to compare the five instance types. The goal is to see which instance type provides the best cost-efficiency. In the second scenario, we create three workloads in which each task of a job has a time limit to obtain its required container. The purpose is to further study if employing a different number of instances affects workload completion rate under the given time limit or not.



(a) Total workload execution time (b) Total instance cost

Fig. 2. The total workload execution time and instance costs of clusters 1 to 5. Note that in Figure 2(a) the first y axis is for the bar plot, whereas the second y axis is for the line plot.

4.1 Scenario 1: Non-urgent Workload

In Scenario 1, the non-urgent workload consists of 100 MapReduce jobs without any time limit to get containers. As a consequence, each task of the jobs in this workload will eventually be executed on a container allocated by RM, implying that the workload completion rate will be 100%. In the workload, half of the jobs are of Type 1 and the other half of the jobs are of Type 2. All the jobs are submitted by a user at the same time, i.e., these jobs have zero inter-arrival time. The purpose is to understand the cost efficiencies of different instance types when all jobs compete for containers simultaneously.

To compare the five instance types in a fair way, we use ABS-YARN to establish five clusters (see Table 3) with the same resource capacity, i.e., 160 vCPUs and 640 GB of memory, and then execute the workload.

Table 3. The specification of the five clusters tested in Scenario 1. Note that the # of vCores and memory are calculated based on Table 1.

Cluster	Instance type	# of instances	# of vCores	Memory (GB)
1	m4.large	80	160	640
2	m4.xlarge	40	160	640
3	m4.2xlarge	20	160	640
4	m4.4xlarge	10	160	640
5	m4.10xlarge	4	160	640

The total workload execution time and the total instance cost spent by the five clusters to execute the non-urgent workload are shown in Figure 2(a) and Figure 2(b), respectively. It is clear that all the clusters have very close workload execution time and total instance costs. Therefore, as shown in Figure 3, they have similar cost-efficiencies. The results confirm that the five instance types have no difference in terms of workload execution performance when the total instance cost is identical. However, when other issues such as management efforts or reliability are further considered, choosing different instance types might result

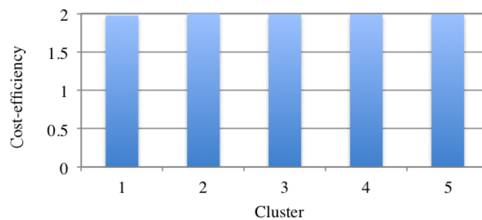


Fig. 3. The cost-efficiencies of clusters 1 to 5.

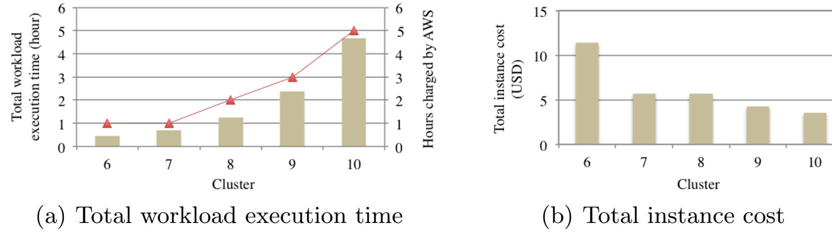
in different effects. However, this depends on users' preferences, and it is outside of the scope of this paper.

We can see from Figure 2(a) that each cluster only spent about 0.44 hours to finish the entire workload, but all of them are charged by AWS for an entire hour (see the line plot in Figure 2(a)). Based on these results, our next question is whether users can find a suitable number of instances to achieve a better cost-efficiency. To answer this, we conduct additional simulations to see how different numbers of the same instance type affect the cost-efficiency. We chose the m4.large instance type as an example and use ABS-YARN to establish another five clusters (see Table 4) and execute the same non-urgent workload.

Table 4. The details of clusters 6 to 10.

Cluster	# of m4.large instances	# of vCores	Memory (GB)
6	80	160	640
7	40	80	320
8	20	40	160
9	10	20	80
10	5	10	40

Figure 4(a) shows that when a cluster employs fewer instances of the same type (implying that total number of available containers is smaller), the time to finish the entire workload increases. Based on the AWS hour-based charge policy, clusters 6 and 7 are charged for 1 hour, cluster 8 is charged for 2 hours, cluster 9 is charged for 3 hours, and cluster 10 is charged for 5 hours. Figure 4(b) illustrates the corresponding total instance cost for each cluster. Naturally, a smaller cluster costs less, but we can see that clusters 7 and 8 have similar total instance costs even though the total number of instances in cluster 7 is twice that of cluster 8. The key reason is that cluster 7 takes 1 AWS hour, but cluster 8 takes 2 AWS hours. By considering both Figure 4(a) and Figure 4(b), we can derive the cost-efficiency of the five clusters (see Figure 5). It is clear that cluster 7 performs the best because it has the highest cost-efficiency among the five clusters. This result is not obvious by just looking at the specifications of AWS instances.



(a) Total workload execution time (b) Total instance cost

Fig. 4. The total workload execution time and instance costs of clusters 6 to 10. Note that in Figure 4(a) the first y axis is for the bar plot, whereas the second y axis is for the line plot.

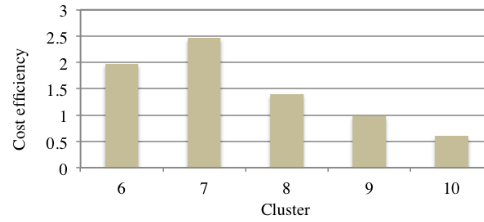


Fig. 5. The cost-efficiencies of clusters 6 to 10.

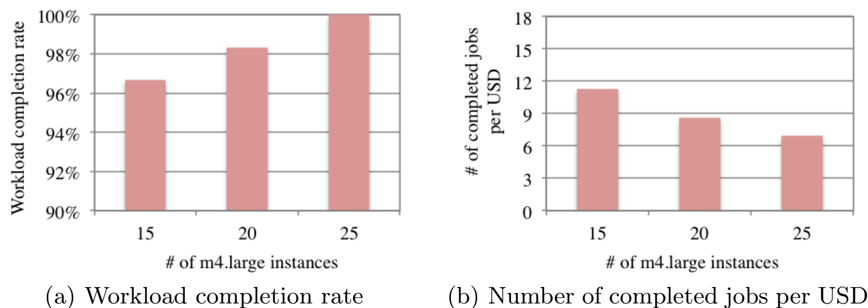
Based on all above simulations, we observe the following phenomenon: First, choosing different AWS m4 instance types makes no difference in terms of performance when the total instance cost is identical. Second, for a particular AWS instance type, employing different numbers of the instance leads to different cost efficiencies.

4.2 Scenario 2: Workload with a Time Limit

Recall that the goal of the second scenario is to investigate if different numbers of instances affect workload completion rate. In this scenario, we continue with the m4.large instance type and create the following three workloads with different job compositions:

- Workload A: 50% of the jobs are Type 1, and 50% of the jobs are Type 2.
- Workload B: 80% of the jobs are Type 1, and 20% of the jobs are Type 2.
- Workload C: 20% of the jobs are Type 1, and 80% of the jobs are Type 2.

All of them have 300 jobs with an inter-arrival pattern following an exponential distribution. The average job inter-arrival time is 158 seconds with the standard deviation of 153 seconds [15]. In addition, the three workloads have the same time limit for each task to get a container. In this experiment, we assume that the time limit is 2 minutes, i.e., each task of a job in the three workloads will be considered as failed if it cannot obtain its desired container within 2 minutes.



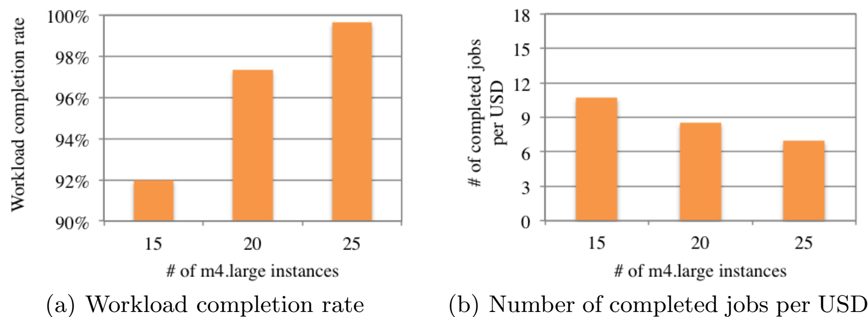
(a) Workload completion rate (b) Number of completed jobs per USD

Fig. 6. The performance results of workload A when different numbers of m4.large instances are employed.

Figure 6(a) shows that when 15 m4.large instances are employed to execute Workload A, the corresponding workload completion rate is 96.7%. When more instances are utilized, the workload completion rate increases. This is because more containers are available to execute the workload. We can see that 25 instances can reach 100% of workload completion rate. However, when the total instance cost is considered, employing 15 instances is the most efficient as this choice offers the highest number of completed jobs per USD (see Figure 6(b)). If a user wants a higher workload completion rate, he/she needs to purchase more instances. Certainly, the cost will increase. However, if workload completion rate has a lower priority than cost, the user can employ fewer instances.

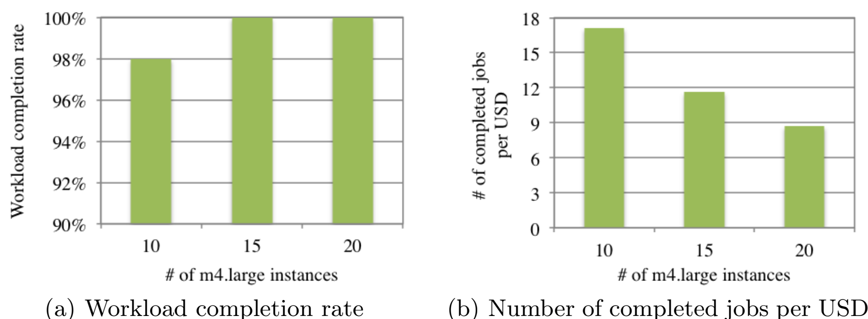
Figure 7(a) and Figure 7(b) show the results when workload B is tested. We can see that the 15 m4.large instances only provide 92% of workload completion rate, and the 25 m4.large instances can no longer complete the entire workload B. The main reason is that workload B has more long jobs than workload A, implying that more containers for workload B are occupied for a longer period than those for workload A. Nevertheless, using 15 m4.large instances still perform the best in terms of the number of completed jobs per USD among the three options. Figure 8(a) and Figure 8(b) show the results when workload C is tested. Different from workloads A and B, the completion rate of workload C is higher on the same number of instances. This is because workload C has more short jobs than the other two. Based on the simulation results of Scenario 2, it is clear that when a workload has different job compositions, employing the same number of AWS instances does not guarantee the same workload completion rate.

Note that our experiments are based on model-based simulations. Hence, the results (e.g., number of completed jobs per USD and workload completion rate) may differ from the actual price and completion. Although there exists some performance deviation [15], with ABS-YARN users can still easily compare different instance deployments and predict the corresponding consequences at the design phase of their applications.



(a) Workload completion rate (b) Number of completed jobs per USD

Fig. 7. The performance results of workload B when different numbers of m4.large instances are employed.



(a) Workload completion rate (b) Number of completed jobs per USD

Fig. 8. The performance results of workload C when different numbers of m4.large instances are employed.

5 Related Work

Many researchers have conducted performance studies by running their scientific applications in cloud environments. For examples, Hazelhurst [11] evaluated the performance of a bioinformatics application and Ramakrishnan et al. [19] studied the performances of e-Science applications in cloud computing. Several works have been introduced to study Amazon Elastic Compute Cloud (Amazon EC2). Napper et al. [17] analyzed the performance of the Linpack benchmarks on different EC2 instance types. Osterman et al. [18] conducted a variety of micro-benchmarks on Amazon EC2.

Garfinkel [9] presented a comprehensive introduction of Amazon's EC2, Simple Storage Service (S3), and Simple Queue Service (SQS) in their securities, privacy controls, legal issues, API limitations, and pricing models. Furthermore, a series of experiments are also provided to study the performances of EC2, S3, and SQS, including average daily throughput, average daily transaction, bandwidth speed, and availability. Jackson et al. [12] performed a comprehensive

evaluation for comparing Amazon EC2 and two conventional HPC platforms (Franklin and Lawrencium) by using real application workloads.

Different from all previous work, the comparison of deployment decisions conducted in this paper is based on a model-centric approach to determine an appropriate deployment configuration at the design time. We compared different deployment decisions by using ABS-YARN [15]. In addition, our comparison focused on several instance types provided by AWS and their tradeoffs, rather than comparing different cloud or HPC platforms.

Stantchev [20] presents a methodology for performance evaluation of cloud computing configurations. The methodology consists of five steps: identify benchmark, identify configuration, run tests, analyze, and recommend. The author conducted a real experiment evaluation by deploying several configurations of a web service benchmark in Amazon EC2, and focused on analyzing the results of two nonfunctional properties: transactions rate and response time. Although this methodology can help users to compare different deployment decisions, the users have to conduct such comparisons in real cloud environments. In other words, the cost to purchase resource instances cannot be reduced.

6 Conclusions

In this paper, we have introduced different instance purchasing options provided by AWS, and also studied different AWS deployments using ABS-YARN. We considered two workload scenarios. In the first scenario, we created a non-urgent workload consisting of 100 MapReduce jobs with the same submission time. The goal was to investigate which AWS instance provides the best cost-efficiency. In the second scenario, we designed additional workloads to further study whether employing different numbers of instances affects workload completion rate.

The results demonstrated that AWS provides a fair instance pricing, i.e., the instance cost doubles when the performance of an instance doubles. Hence, from a performance perspective, choosing a better or worse instance makes no difference when the total instance cost is fixed. However, this may not be the case when other issues such as management efforts, reliability, and data locality are further considered. Furthermore, purchasing different numbers of a particular instance type may lead to different workload execution time and instance costs. Our results in the second scenario also showed that purchasing different numbers of the same AWS instances significantly affected workload completion rate, especially when workloads consist of different compositions of long jobs and short jobs.

In general, if users want to achieve a high cost-efficiency and a high workload completion rate, they need to conduct multiple tuning and comparisons between different instance options. With ABS-YARN, they can easily evaluate and compare different deployment decisions with less cost and effort.

References

1. Amazon EC2 FAQs. Q: What is an "EC2 compute unit" and why did you introduce it? https://aws.amazon.com/ec2/faqs/#hardware-information. Accessed 27. April 2016.
2. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/?nc1=h_ls.
3. Apache Hadoop. http://hadoop.apache.org/.
4. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
5. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
6. J. Bort. Amazon still dominates the $16 billion cloud market. *UK Business Insider*, http://uk.businessinsider.com/synergy-research-amazon-dominates-16-billion-cloud-market-2015-2, Feb. 2015.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
8. enwiki. http://dumps.wikimedia.org/enwiki/.
9. S. L. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3, and SQS. Technical Report TR-08-07, Center for Research on Computation and Society School for Engineering and Applied sciences, Harvard University, Aug. 2007. Available via web: https://dash.harvard.edu/handle/1/24829568.
10. R. Hähnle and E. B. Johnsen. Designing resource-aware cloud applications. *IEEE Computer*, 48(6):72–75, 2015.
11. S. Hazelhurst. Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, SAICSIT '08, pages 94–103. ACM, 2008.
12. K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '10, pages 159–168. IEEE, 2010.
13. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
14. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.
15. J.-C. Lin, I. C. Yu, E. B. Johnsen, and M.-C. Lee. ABS-YARN: A formal framework for modeling Hadoop YARN clusters. In P. Stevens and A. Wasowski, editors,

19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016), volume 9633 of *Lecture Notes in Computer Science*. Springer, 2016.

16. A. Murthy, V. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 2014.
17. J. Napper and P. Bientinesi. Can cloud computing reach the top500? In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, UCHPC-MAW '09, pages 17–20. ACM, 2009.
18. S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. An early performance analysis of cloud computing services for scientific computing. Technical Report PDS-2008-006, Delft University of Technology, Dec. 2008. Available via web: `http://www.ds.ewi.tudelft.nl/reports/2008/PDS-2008-006.pdf`.
19. L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining future platform requirements for e-science clouds. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 101–106. ACM, 2010.
20. V. Stantchev. Performance evaluation of cloud computing offerings. In *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, ADVCOMP '09, pages 187–192. IEEE, 2009.
21. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In G. M. Lohman, editor, *ACM Symposium on Cloud Computing (SOCC'13)*, pages 5:1–5:16, 2013.