A Workflow Scheduling Technique Using Genetic Algorithm in Spot Instance-Based Cloud

Daeyong Jung[1], Taeweon Suh[1], Heonchang Yu[1] and JoonMin Gil[2*]
[1] Dept. of Computer Science Education, Korea University
Seoul, Korea
[e-mail: {karat, suhtw, yuhc}@korea.ac.kr]
[2] School of Information Technology Engineering, Catholic University of Daegu
Daegu, Korea
[e-mail: jmgil@cu.ac.kr]
*Corresponding author: JoonMin Gil

Received December 2, 2013; revised May 10, 2014; revised July 9, 2014; accepted August 19, 2014; published September 30, 2014

Abstract

Cloud computing is a computing paradigm in which users can rent computing resources from service providers according to their requirements. A spot instance in cloud computing helps a user to obtain resources at a lower cost. However, a crucial weakness of spot instances is that the resources can be unreliable anytime due to the fluctuation of instance prices, resulting in increasing the failure time of users' job. In this paper, we propose a Genetic Algorithm (GA)-based workflow scheduling scheme that can find the optimal task size of each instance in a spot instance-based cloud computing environment without increasing users' budgets. Our scheme reduces total task execution time even if an out-of-bid situation occurs in an instance. The simulation results, based on a before-and-after GA comparison, reveal that our scheme achieves performance improvements in terms of reducing the task execution time on average by 7.06%. Additionally, the cost in our scheme is similar to that when GA is not applied. Therefore, our scheme can achieve better performance than the existing scheme, by optimizing the task size allocated to each available instance throughout the evolutionary process of GA.

Keywords: Cloud computing, Spot instances, Workflow, Price history, Fault tolerance, Genetic algorithm

http://dx.doi.org/10.3837/tiis.2014.09.010

1. Introduction

In recent years, due to the increased interest in cloud computing, many cloud projects and commercial systems such as the Amazon Elastic Compute Cloud (EC2) [1], and FlexiScale [2] have been implemented. Cloud computing provides high utilization and high flexibility for managing computing resources. In addition, cloud computing services provide a high level of scalability of computing resources combined with Internet technology that are distributed among several customers [3, 4, 5]. In most cloud services, the concept of an instance unit is used to provide users with resources in a cost-efficient manner. Generally, instances are classified into three types: on-demand instances, reserved instances, and spot instances. On-demand instances allow a user to pay for computing capacity by the hour, with no long-term commitments. This frees users from the costs and complexities of planning, purchasing, and maintaining hardware, and transforms what are usually large fixed costs into much smaller variable costs [1]. Reserved instances allow a user to make a low, one-time payment to reserve instance capacity and further reduce the user's cost. While in reserved instances a user pays a yearly fee and receives a discount on hourly rates, in on-demand instances a user pays one hourly rate [6]. On the other hand, spot instances allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current spot price. Based on the supply and demand of spot instances, the spot price is changed. Customers whose bid exceeds the spot price can gain access to the available spot instances. If the applications executed are time-flexible, spot instances can significantly decrease the Amazon EC2 costs [7]. Therefore, spot instances may incur lower costs while performing tasks than on-demand instances.

Spot market-based cloud environment configures the spot instance. The environment affects the successful completion or failure of tasks depending on the changing spot prices. Because spot prices have a market structure and follow the law of demand and supply, cloud services in Amazon EC2 can provide a spot instance when a user's bid is higher than the current spot price. Furthermore, a running instance stops when a user's bid becomes less than or equal to the current spot price. After a running instance stops, it restarts when a user's bid becomes greater than the current spot price [8].

Scientific applications, in particular, make the current common of workflow. However, spot instance-based cloud computing has variable performance, because the available execution time of spot instances depends on the spot price. The completion time for the same amount of a task varies according to the performance of an instance. In other words, the failure time of each instance differs according to the user's bid and the performance in an instance. Thereby, we infer that the completion time of a task in an instance increases when a failure occurs. For efficient execution of tasks, we analyze the task and instance information from the price history data of spot instances, and estimate the task size and instance availability from the analyzed data. A workflow is created based on each available instance and the task size. However, the created workflow has a problem in that it does not consider the failure time of each instance. To solve this problem, we propose a scheme to change the task size of each instance using a Genetic Algorithm (GA). Our proposed scheduling scheme uses workflow mechanisms and GA to handle job execution. In our scheme, a user's job is executed within selected instances and the user's budget is stretched.

2. Related Works

Two different environments in cloud computing have been considered in recent research studies: reliable environments (with on-demand instances [9, 10]) and unreliable environments (with spot instances [8, 11, 12, 12]). Our study falls into the latter category of cloud computing environments.

Typically, studies on spot instances have mainly focused on performing tasks at low monetary costs. The spot instances in the Amazon EC2 offer cloud services to users at lower prices at the expense of reliability [1]. Cloud exchange [14] provides the actual price history of EC2 spot instances. There has been numerous studies on resource allocation [12, 13], fault tolerance [8, 11, 12, 15], workflow scheduling [16, 17], and the use of Genetic Algorithms [18, 19, 20] for spot instances.

In the field of resource allocation, Voorsluys et al. [12] provided a solution for running computation-intensive tasks in a pool of intermittent VMs. To mitigate potential unavailability periods, the study proposed a multifaceted fault-aware resource provisioning policy. Voorsluys et al's solution employed price and runtime estimation mechanisms. The proposed strategy achieved cost savings and stricter adherence to deadlines. Zhang et al. [13] demonstrated how best to match customer demand in terms of both supply and price, and how to maximize provider revenue and customer satisfaction in terms of VM scheduling. Their model was designed to solve the problem of discrete-time optimal control. It achieved higher revenues than static allocation strategies and minimized the average request waiting time. Our work differs from [12] and [13] in that we focus on reducing the rollback time after a task failure, achieving cost savings and reducing the total execution time.

In the fault tolerance category, two similar studies ([11] and [8]) proposed enforcing fault tolerance in cloud computing with spot instances. Based on the actual price history of EC2 spot instances, these studies compared several adaptive checkpointing schemes in terms of monetary costs and job execution time. Goiri et al. [10] evaluated three fault tolerance schemes - checkpointing, migration, and job duplication - assuming fixed communication costs. The migration-based scheme showed a better performance than the checkpointing or the job duplication-based scheme. Voorsluys et al. [12] also analyzed and evaluated the impact of checkpointing and migration on fault tolerance using spot instances. Our paper differs from [8, 10, 11, 12] in that we utilize two thresholds for fault tolerance.

A workflow is a model that represents complex problems with structures such as Directed Acyclic Graphs (DAG). Workflow scheduling is a kind of global task scheduling as it focuses on mapping and managing the execution of interdependent tasks on shared resources. The existing workflow scheduling methods have limited scalability and are based on centralized scheduling algorithms. Consequently, these methods are not suitable for spot instance-based cloud computing. In spot instances, the available time and instance costs have to be considered for job execution. Fully decentralized workflow scheduling systems use a chemistry-inspired model to determine the instance in a community cloud platform [16]. The throughput maximization strategy is designed for transaction-intensive workflow scheduling that does not support multiple workflows [17]. Our proposed scheduling scheme guarantees an equal task distribution across available instances in spot instance-based cloud computing.

A GA is a popular search technique that uses the concept of evolution in the natural world to solve optimization problems [18, 19, 20]. It has been used for cloud scheduling in various studies [21, 22, 23]. The synapsing variable-length crossover (SVLC) algorithm provides a biologically inspired method for performing meaningful crossover between variable-length genomes [18]. However, traditional GAs [19, 20] operate on a population of fixed-length genomes. In additional, those have the problem to relate a set of potential solutions. Our GA utilizes the crossover to adopt the variable-length. In [21], the scheduling of VM resources in a load balanced manner was based on GA. In a private cloud environment, a Modified Genetic algorithm (MGA) was utilized for task scheduling with a combination of Shortest Cloudlet to Fastest Processor (SCFP) and Longest Cloudlet to Fastest Processor (LFCP) methods [22]. In this study, authors also used a meta-heuristic GA as an optimization method. Fatma et al. [23] adapted task scheduling to use two fitness functions. The first fitness function is concerned with minimizing the total execution time, and the second is concerned with the load balance. However, the existing studies did not consider the failure time of instances that makes task execution stop. Our GA-based scheduling method considers the failure time of instances, the task execution time, and task execution costs.

3. System Architecture

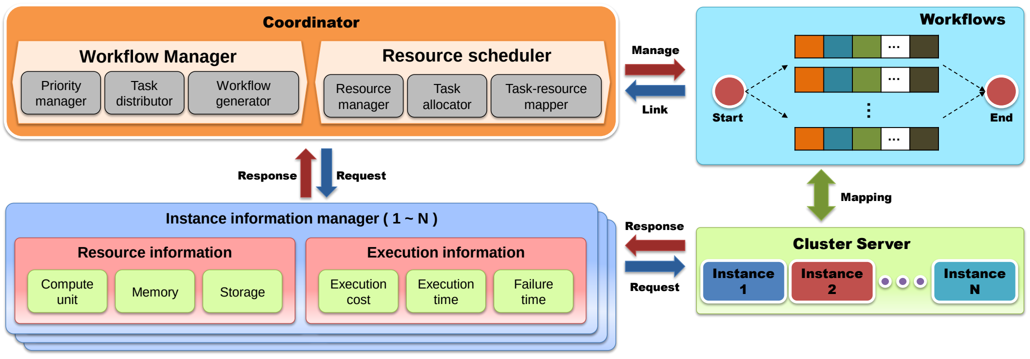


Fig. 1. Mapping between workflows and instances

Our proposed scheme expands on our previous work [24] and includes a workflow scheduling algorithm. We make workflow tasks based on available instances and propose how to handle tasks. Each available instance operates workflow tasks and uses checkpointing scheme in Section 3.3. **Fig. 1** presents the mapping relationship between workflows and instances, and illustrates the roles of the instance information manager, the workflow manager, and the resource scheduler. The instance information manager obtains the information required for job allocation and resource management. This information includes the VM specifications for each instance and execution related information such as the execution costs, execution completion time, and failure time. The execution-related information is calculated by using the selected VM and is based on the spot history. The workflow manager and resource scheduler extract the necessary execution-related information from the instance information manager. First, the workflow manager generates the workflow for the request job. The generated workflow determines the task size according to the VM performance, the execution time and costs, and the failure time in the selected instance. Second, the resource scheduler manages the resource and allocates the task to handle the job. Resource and task managements perform reallocation when the resource cannot get the mapping-related information for the task, or when the task has a fault during execution.

In the above model, our proposed scheme uses the workflow in spot instances to minimize job processing time within the user's budget. The task size is determined by evaluating the availability and performance of each instance in order to minimize the job processing time. The available time is estimated by calculating the execution time and cost using the price history of spot instances. This helps to improve the performance and stability of task processing. The estimated time data is used in assigning the amount of task to each instance. Our proposed scheme reduces out-of-bid situations and improves the job execution time. However, the total cost is higher than the cost when workflow scheduling is not used.

3.1 Instance types

An instance means a VM rented by a cloud user. Instances are classified into two types: on-demand instances and spot instances. In on-demand instances, users can use VM resources after paying a fixed cost on an hourly basis. On the other hand, when using spot instances, users can use VM resources only when the price of the instances is lower than their bid. The difference between the two instance types is as follows. In on-demand instances, a failure does not occur during task execution, but the cost is comparatively high. In contrast, the cost of spot instances is lower than that of on-demand instances. However, in the case of spot instances, there is a risk of task failures when the price of the instance becomes higher than the user's bid.

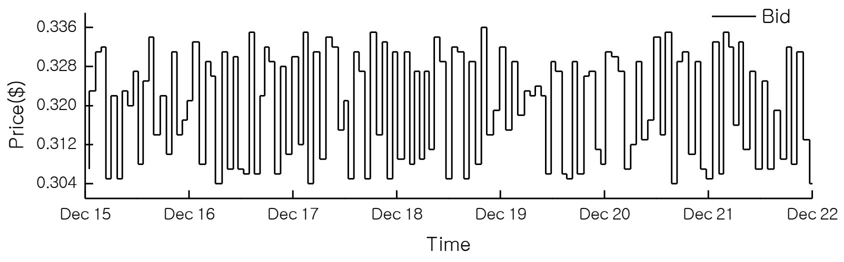


Fig. 2. Price history of EC2's spot instances for c1-xlarge

Amazon allows users to bid on unused EC2 capacity that includes 42 types of spot instances [1]. Their prices, which are referred to as spot prices, change dynamically based on the supply and demand. **Fig. 2** shows an example of spot price fluctuation during seven days in December 2010 for c1-xlarge (High-CPU Spot Instances - Extra Large) [14]. Our proposed system model is based on the characteristics of Amazon EC2's spot instances, which are as follows:

- The system provides a spot instance when a user bid is higher than the current price.
- The system immediately stops the spot instance, without giving any notice, when a user bid becomes less than or equal to the current price. We refer to this situation as an out-of-bid event or a failure.
- The system does not charge for the last partial hour when the system stops the spot instance.
- The system charges for the last partial hour when the user voluntarily terminates the spot instance.
- The system provides the spot price history.

3.2 Estimated Interval

Our checkpointing operation is performed by analyzing the price variations during selected time intervals in the past. We estimate the job execution time and cost from the analyzed data. These estimations are combined with the failure probability in order to calculate the thresholds for the checkpointing operation. The proper estimation of the execution time and cost is crucial for maintaining the credibility of service providers with customers.

In this paper, we introduce a terminology referred to as Estimated Interval (EI). **Fig. 3** shows an illustrative definition of the EI. The detailed definition is as follows:

- Pure task time: The time taken to execute a task on a selected instance when there are no failures.
- Past pure task time: The sum of the time taken for task execution on the selected instance in the past, excluding failure times. The time information is extracted from the price history.
- Past failure time: The sum of failure times for task execution in the past. A failure occurs when the current user bid goes below the past spot price.
- Estimated interval (EI): The sum of the past pure task time and the past failure time.
- Expected cost: The average of costs charged for task execution in EIs.

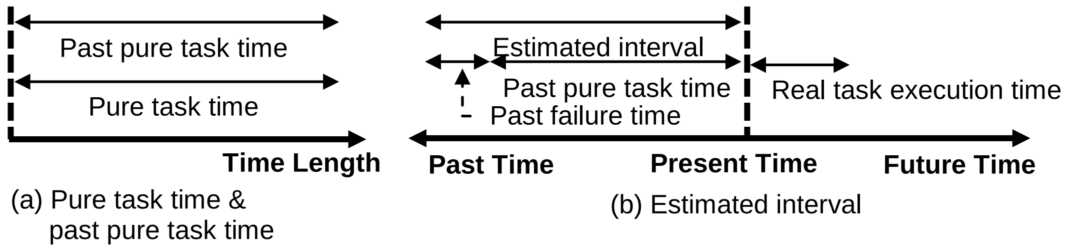


Fig. 3. EI illustration

3.3 Our Proposed Checkpointing Scheme

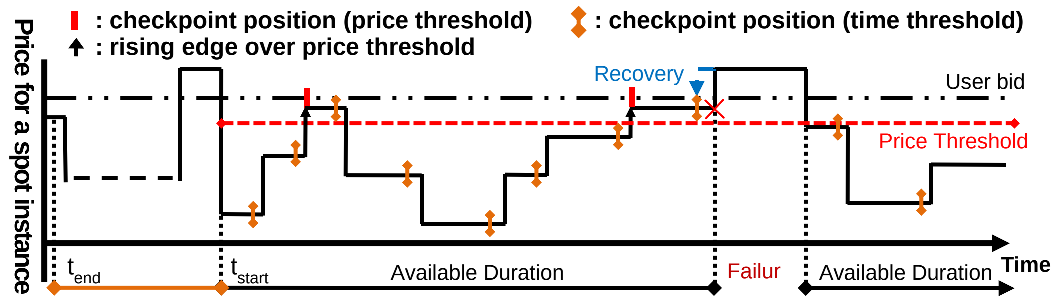

: checkpoint position (price threshold)
: checkpoint position (time threshold)
: rising edge over price threshold
Price for a spot instance
Recovery
User bid
Price Threshold
t_end
t_start
Available Duration
Failur
Available Duration
Time

Fig. 4. Our proposed checkpointing scheme

Fig. 4 illustrates our proposed checkpointing scheme [24]. This scheme performs checkpointing using two kinds of thresholds - price threshold and time threshold - depending on the expected execution time based on the price history. Let t_{start} and t_{end} denote a start point and an end point in the expected execution time, respectively. Based on t_{start} and t_{end} , we obtain the price threshold ($PriceTh$) and the time threshold ($TimeTh_{P_i}$), which are used as thresholds in our proposed checkpoint scheme. The price threshold, $PriceTh$, can be calculated as

$$PriceTh = \frac{P_{\min} + User_{bid}}{2} \tag{1}$$

where $User_{bid}$ represents the bid suggested by the user. $P_{\min}$ is the available minimum price that the function *PriceMin* extracts in the period between t_{start} and t_{end}. This is given as follows:

$$P_{\min} = PriceMin(t_{start}, t_{end}) \tag{2}$$

The time threshold of the price P_i, $TimeTh_{P_i}$, can be calculated as

$$TimeTh_{P_i} = AvgTime_{P_i}(t_{start}, t_{end}) \times (1 - F_{P_i}) \tag{3}$$

where F_{P_i} is the failure probability of price P_i, and $AvgTime_{P_i}(t_{start}, t_{end})$ represents the average execution time of P_i in the period between t_{start} and t_{end}.

4. Genetic Algorithm Based Job Scheduling

The main goal of our scheduling method is to minimize the execution time and the cost of running applications. Our scheduling method is based on the genetic algorithm (GA) that is a well-known and robust search technique for solving large-scale optimization problems. The GA consists of the following five steps:
(1) Randomly generate an initial population.
(2) Generate offspring using genetic operators such as crossover and mutation.
(3) Rank chromosomes using the defined fitness function.
(4) Update and evaluate the best-ranked chromosomes based on the selection.
(5) Repeat steps 2–4 if the number of pre-determined constraints cannot satisfy the processing of GA generation.

4.1 Initial Population

Initially, all chromosomes in a population are randomly constructed without any knowledge of experts. Before describing the representation of a chromosome, let us consider the definitions of both instances and jobs that each chromosome consists.

- N : the number of instance types;
- M : the number of jobs;
- $I = \{I_1, I_2, \cdots, I_N\}$: a set of instances;
- I_i : an instance of type i $(1 \le i \le N)$;
- $J = \{J_1, J_2, \cdots, J_M\}$: a set of jobs;
- L_j : the number of tasks in *j*-th job;
- $J_j = \{t_{j,1}, t_{j,2}, \cdots, t_{j,L_j}\}$: a set of tasks in *j*-th job $(1 \le j \le M)$;
- $t_{j,k}$: *k*-th task in *j*-th job $(1 \le k \le L_j)$.

To avoid the complexity of task allocations to instances, we assume two constraints as follows: one is that all tasks have an identical size. The other is that there is no sequence for task execution on an instance. Under this assumption, we construct the structure of two-dimensional chromosomes. **Fig. 5** illustrates the two-dimensional chromosome represented as a grid with N rows and M columns. In this figure, a gene $g_{(I_i, J_j)}$ means the *i*-th row and the *j*-th column $(1 \le i \le N, 1 \le j \le M)$ in the chromosome and is interpreted as follows.

$$g_{(I_i, J_j)} = \begin{cases} l \text{ numbers of tasks in } J_j \text{ are allocated to the instance } I_i, & \text{if } i = 1 \\ (l - k) \text{ numbers of tasks in } J_j \text{ are allocated to the instance } I_i, & \text{if } i > 1 \end{cases} \tag{4}$$

where k and l respectively represent the values of the adjacent two genes, $g_{(I_{i-1},\, J_j)}$ and $g_{(I_i,\, J_j)}$ $(1 \le k \le l \le L_j)$.

To meet the requirements for task execution, n instances satisfying the condition in Eq. (5) are chosen when allocating tasks to initial instances. Our task distribution method determines the task size in order to allocate a task to a selected instance. Based on a compute-unit and an available state, the task size of an instance I_i for J_j ($T_{I_i}^{J_j}$) is calculated as

$$T_{I_i}^{J_j} = \left(\frac{U_{I_i} \times A_{I_i}}{\sum_{i=1}^{N} (U_{I_i} \times A_{I_i})} \right) \times \frac{1}{U_{I_i}} \times T_{request}^{J_j} \times U_{baseline} \tag{5}$$

where $T_{request}^{J_j}$ represents the total size of tasks in J_j required for executing a user request. In an instance I_i, U_{I_i} and A_{I_i} represent the compute-unit (the product of CPU and cores) and the available state, respectively. The available state A_{I_i} can be either 0 (unavailable) or 1 (available). The $U_{baseline}$ represents the compute-unit of the selected instance to request the user. The task size is decided by the compute-unit rate based on the baseline.

Each gene $g_{(I_i,\, J_j)}$ shows the number of task sequences according to the compute-unit of the selected instance and the variation of task size. The task sequences are related to the task size. Eq. (6) represents the number of task sequences in gene $g_{(I_i,\, J_j)}$.

$$g_{(I_i,\, J_j)} = \begin{cases} (T_{I_i}^{J_j})', & \text{if } i = 1 \\ g_{(I_{i-1},\, J_{j-1})} + (T_{I_i}^{J_j})', & \text{if } i > 1 \end{cases} \tag{6}$$

The task size of each instance in the chromosome adjusts based on Eq. (7). The variation of task size $\alpha_{I_i}^{J_j}$ is randomly set by Eq. (8).

$$(T_{I_i}^{J_j})' = T_{I_i}^{J_j} + \alpha_{I_i}^{J_j} \tag{7}$$

$$\alpha_{I_i}^{J_j} = [-T_{I_i}^{J_j},\, T_{I_i}^{J_j}] \tag{8}$$

When an initial population is constructed, all genes in the chromosome are randomly set to an integer value adopting Eq. (6).

	J_1	J_2	J_3	$\cdots$	J_j	$\cdots$	J_M
I_1	5	3	5		$(T_{I_1}^{J_j})'$		11
I_2	10	6	10		$g_{(I_1,J_j)} + (T_{I_2}^{J_j})'$		22
I_3	15	9	30		$g_{(I_2,J_j)} + (T_{I_3}^{J_j})'$		33
$\vdots$							
I_N	40	30	50		$g_{(I_{N-1},J_j)} + (T_{I_N}^{J_j} + \alpha_{I_N}^{J_j})'$		60

Fig. 5. Chromosome representation

4.2 Fitness Function

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation of the population. Additionally, the fitness function is the criterion for determining the quality of chromosomes in the population. It directly reflects the performance of task distribution. In our paper, the standard deviation is used as the fitness function to evaluate the performance of chromosomes. Based on the standard deviation for each chromosome C_k, our fitness function $f(\sigma_{C_k})$ is defined as follows.

$$f(\sigma_{C_k}) = 1 \Big/ \left(1 + \sqrt{\left(\sum_{i=1}^{N} (EET_{I_i} - Avg_{C_k})^2\right) / N}\right) \tag{9}$$

In Eq. (9), EET_{I_i} is the total estimated execution time of the instance I_i. Avg_{C_k} is the average of total estimated execution time and can be calculated as

$$Avg_{C_k} = \frac{\sum_{i=1}^{N} EET_{I_i}}{N}, 1 \le k \le P \tag{10}$$

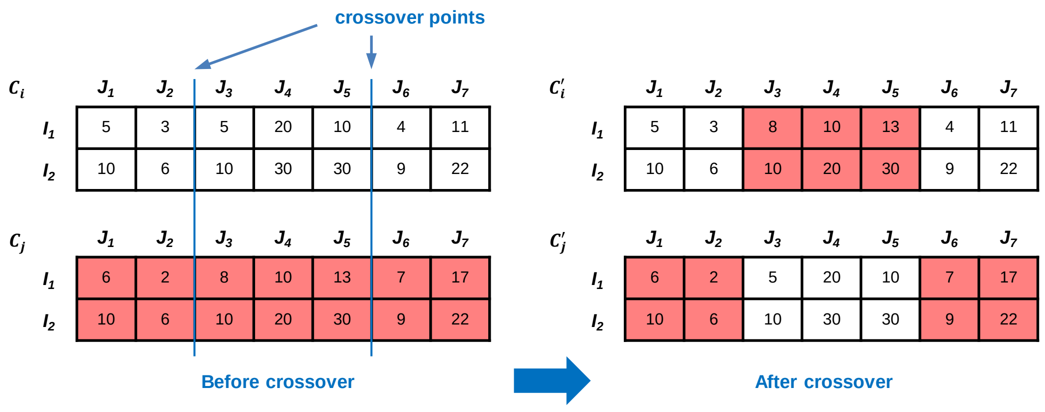
where, P represents population size.

4.3 Selection

The fittest chromosomes have higher probability to be selected for the next generation. We use two selection mechanisms. One is the elitism method that forces the GA to retain some number of the best chromosomes at each generation. The other is a roulette wheel method. Given the population size P, P - 1 chromosomes in a new population are made by the roulette wheel method. The elitism method is applied to produce the remaining one chromosome, which is the best of chromosomes in population pool.

4.4 Genetic Operations (Crossover, Mutation, and Reproduction)

The successive generation in GA is determined by a set of operators that recombine and mutate selected members of the current population. The crossover operator produces two new offsprings from two parents, leading to improving the fitness of chromosomes in the current population. We adapt two-point crossover method to the spot instance environment. In the typical two-point crossover, a pair of chromosomes is selected according to the crossover probability p_c, and two crossover points are randomly selected. **Fig. 6** illustrates an example of two-point crossover operation in our genetic algorithm.



C_i	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	5	3	5	20	10	4	11
I_2	10	6	10	30	30	9	22

C_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	6	2	8	10	13	7	17
I_2	10	6	10	20	30	9	22

C_i'	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	5	3	8	10	13	4	11
I_2	10	6	10	20	30	9	22

C_j'	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	6	2	5	20	10	7	17
I_2	10	6	10	30	30	9	22

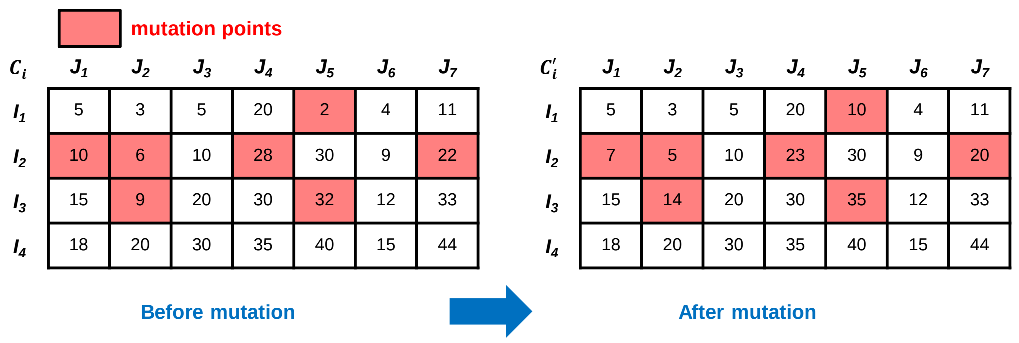
Fig. 6. Example of two-point crossover

The mutation operation changes the value of genes in the chromosome according to the mutation probability p_m. Our mutation operation is depicted in **Fig. 7**. The number of mutation points is randomly selected by row. After selecting the number of rows, the number of column mutations is randomly selected in each row. After the mutation operation is applied, the gene $g'_{(I_i,J_j)}$ is calculated by

$$g'_{(I_i,J_j)} = \begin{cases} g_{(I_i,J_j)} + r_1, \text{ if } \; Avg_{C_k} > EET_{I_i} \\ g_{(I_i,J_j)} - r_2, \text{if } \; Avg_{C_k} \le EET_{I_i} \end{cases} \tag{11}$$

where $g_{(I_i,J_j)}$ is a previous gene in the chromosome C_k. r_1 and r_2 represent the variation to the gene $g_{(I_i,J_j)}$. There are two cases. In the first case (i.e. r_1), if the EET_{I_i} of the instance I_i is given less than the average of *EET* of available instances, the gene $g_{(I_i,J_j)}$ is changed into the new gene $g'_{(I_i,J_j)}$ greater than the current value. The r_1 is extracted from the value ranging between $g_{(I_i,J_j)}$ and $g_{(I_{i+1},J_j)}$. In the second case (i.e. r_2), if the EET_{I_i} of the instance I_i is greater than the average of *EET* of available instances, the gene $g_{(I_i,J_j)}$ is changed into the new gene value less than the current value. The r_2 is extracted from the value ranging between $g_{(I_{i-1},J_j)}$ and $g_{(I_i,J_j)}$.

The above-mentioned two cases are applied to only instances between I_1 and I_{N-1}. The genes of last instance ($g_{(I_N,J_j)}$) can not be changed, because they have the final task number in a job; i.e, the mutation operation is not applied to these genes.


🟥 mutation points

C_i	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	5	3	5	20	**2**	4	11
I_2	**10**	**6**	10	**28**	30	9	**22**
I_3	15	**9**	20	30	**32**	12	33
I_4	18	20	30	35	40	15	44

Before mutation

C'_i	J_1	J_2	J_3	J_4	J_5	J_6	J_7
I_1	5	3	5	20	**10**	4	11
I_2	**7**	**5**	10	**23**	30	9	**20**
I_3	15	**14**	20	30	**35**	12	33
I_4	18	20	30	35	40	15	44

After mutation

Fig. 7. Example of mutation

The reproduction operation is used to obtain the modifed chromosome for the next generation. The reproducted chromosomes selects chromosomes to consider the low ranking of the fitness according to the reproduction probability p_r. Each chromosome calculates the ranking of the fitness based on Eq. (9). The selected chromosome deletes current chromosome and than reproduces new chromosome. The selected chromosome adjusts the task size based on the elitism chromosome of the selection phase. The task size of each instance is determined according to the standard deviation of each instance in the elitism chromosome.

4.5 GA-based Workflow Scheduling (GAWS) Algorithm

Our GA-based Workflow Scheduling (GAWS) algorithm is described in **Fig. 8**. The Task_flag represents the occurrence of a task execution, and its initial value is false. When the task execution is normal (i.e., Task_flag is true), the scheduler performs a workflow operation (lines 3-27). The GA_flag represents the execution of GA, and its initial value is true; Lines 17-24 are initial population, fitness function, and genetic operations of GA, respectively. Lines 28-35 show the workflow function.

```
     /*Initialization */
 1:  Boolean Task_flag = false // a flag representing occurrence of a task execution
 2:  Boolean GA_flag = true // a flag representing occurrence of GA
 3:  while (search user's job) do
 4:      if  (require job execution by the user) then
 5:          take the cost and total execution time by the user;
 6:          Task_flag = true;
 7:      end if
 8:      if (job_flag) then
 9:          forall instance $I_i$ ∈ all Instances do // search available instances
10:              retrieve an instance information to meet the user's requirement in an instance $I_i$;
11:              analyze an available execution time and cost in an instance $I_i$;
12:              store the analyzed available instance to a $\text{queue}_{\text{instance}}$;
13:          end forall
14:          forall instance $I_i$ ∈ $\text{queue}_{\text{instance}}$ do // operate an initial population
15:              the task size of an instance $I_i$ is calculated as Eq. (6);
16:          end forall
             /* perform GA operation */
17:          while(GA_flag) do
                 /* perform fitness functions */
18:              calculate the standard deviation of each chromosome $C_k$ , $\sigma_{C_k}$ as Eq. (9);
                 /*  perform  selection (elitism and roulette method) */
19:              chose best chromosome at each generation;
20:              other chromosomes calculate the probability to select genetic operations;
                 /*  perform  crossover, mutation, and  reproduction based on fitness functions */
21:              perform  two-point crossover to select a pair of chromosomes as the crossover
                 probability $p_c$ ;
22:              switches the value to select genes with the mutation probability  $p_m$ ;
23:              perform  reproduction  as the reproduction  probability  $p_r$ ;
24:          end while
25:          invoke workflow ( );
26:      end if
27:  end while
28:  Thread_Function workflow ( ) begin
29:      while (task execution does not finish) do
30:          calculate on priority list for the priority job allocation;
31:          forall instance  $I_i$ ∈ $\text{queue}_{\text{instance}}$ do
32:              allocate tasks to the instance $I_i$;
33:          end forall
34:      end while
35:  end  Thread_Function
```

Fig. 8. GAWS algorithm

5. Performance Evaluation

In this section, we evaluate and analyze the performance of the proposed scheme through simulations. The performance comparision for the scheme is also presented.

5.1 Simulation Environment

Before performing our simulations, we describe spot-instance environments and the simulation parameters used in our GA. Our simulations were conducted using the history data obtained from Amazon EC2 spot instances [14]. The history data before 10-01-2010 was used to extract the expected execution time and failure occurrence probability for our checkpointing scheme. The applicability of our scheme was tested using the history data after 10-01-2010.

Table 1. Resource type information

Instance type name	**Compute unit**	**Virtual cores**	**Spot price min**	**Spot price average**	**Spot price max**
m1.small (Standard)	1 EC2	1 core (1 EC2)	\$0.038	\$0.040	\$0.053
m1.large (Standard)	4 EC2	2 cores (2 EC2)	\$0.152	\$0.160	\$0.168
m1,xlarge (Standard)	8 EC2	4 cores (2 EC2)	\$0.076	\$0.080	\$0.084
c1.medium (High-CPU)	5 EC2	2 cores (2.5 EC2)	\$0.304	\$0.323	\$1.52
c1.xlarge (High-CPU)	20 EC2	8 cores (2.5 EC2)	\$0.532	\$0.561	\$0.588
m2.xlarge (High-Memory)	6.5 EC2	2 cores (3.25 EC2)	\$0.532	\$0.561	\$0.588
m2.2xlarge (High-Memory)	13 EC2	4 cores (3.25 EC2)	\$0.532	\$0.561	\$0.588
m2.4xlarge (High-Memory)	26 EC2	8 cores (3.25 EC2)	\$1.064	\$1.22	\$1.176

Table 2. Parameters and values for simulation

Simulation parameter	**Task time interval**	**Baseline**	**Distribution time**	**Merge time**	**Checkpoint time**	**Recovery time**
Value	43,200(s)	m1.xlarge	300(s)	300(s)	300(s)	300(s)

Table 3. Parameters and values for GA

Simulation parameter	**Value**
Population size	50
Number of generations	100
Crossover probability (p_c)	0.3
Mutation probability (p_m)	0.1
Reproduction probability (p_r)	0.3

In the simulations, one type of spot instance was applied to demonstrate the effect of task execution time analysis on the performance. Table 1 shows the various resource types used in Amazon EC2. In this table, resource types are comprised of different instance types. First, standard instances offer a basic resource type. Second, high-CPU instances offer more compute units than other resources, and can be used for computation-intensive applications. Finally, high-memory instances offer more memory capacity than other resources, and can be used for high-throughput applications such as databases and memory caching applications. In the simulation environments, we compared the performance of our proposed scheme with that of the existing schemes without considering task distribution based on the task execution time.

Table 1 shows the information related to resource types in each instance, and **Table 2** shows the parameters and values for our simulation. **Table 3** shows the parameters and values for our genetic algorithm. The spot price information was extracted from the spot history data from 11-30-2009 to 01-23-2011. The user's bid was taken as the average of the spot prices from the spot history data. Using Eq. (5), the task size was decided by the compute-unit rate based on the baseline m1.xlarge.

5.2 Effect of GA Analysis

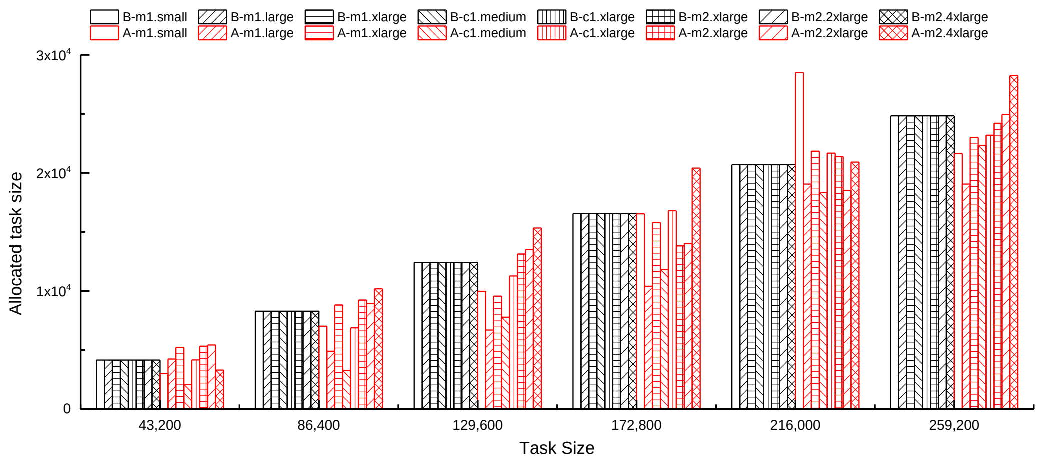


Fig. 9. Size variations in requested tasks

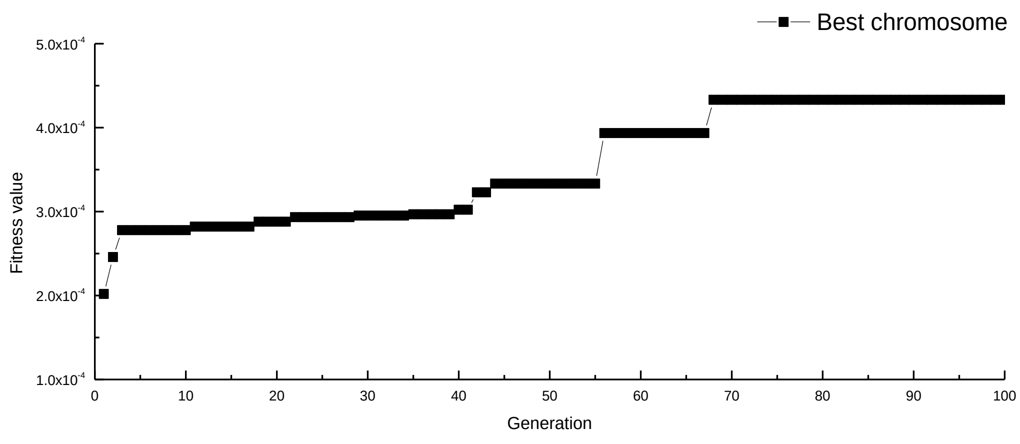


Fig. 10. Fitness curve

In this section, we show the size variations of requested task and the fitness variation according to GA. **Fig. 9** shows the size variations of requested tasks in each instance before and after using GA. In this figure, "B" and "A" denote the before and after GA, respectively, and each instance type (m1.small, m1.large, etc.) indicates the task size allocated to each instance among the requested tasks of users. As shown in **Fig. 9**, when GA is applied, the task size is varied for each instance. This variation directly relates to the failure time of each instance; i.e, it is because more tasks are allocated to instances with low failure time by the evolutionary process of GA.

Fig. 10 shows the fitness curve of the best chromosome at each generation when GA is applied in case that the task size is 259,200. In this figure, we can observe that the value of fitness becomes higher as the generation increases. Therefore, we can see that our genetic algorithm-based scheme can sufficiently find the optimum solution.

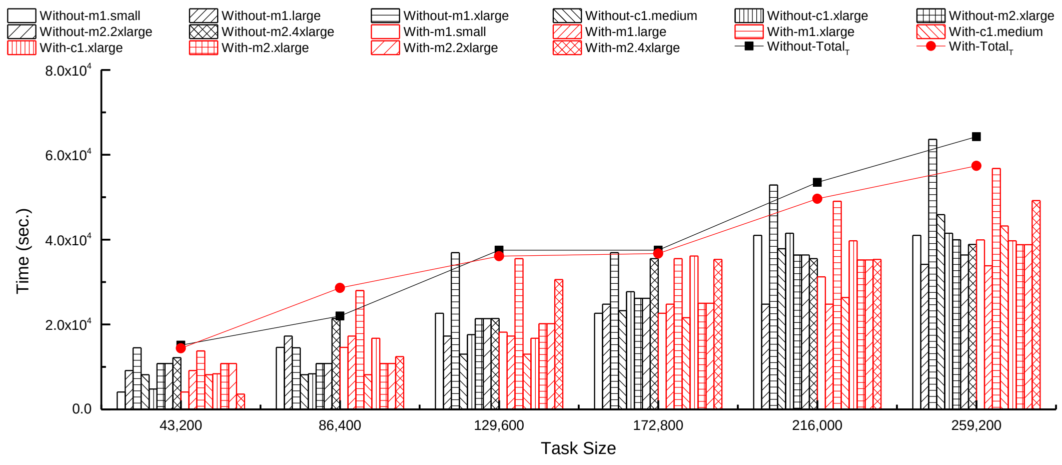


Fig. 11. Comparison with and without crossover operation

Fig. 11 shows the performance comparison between with and without crossover operation. In this figure, "Without" and "With" denote without and with the crossover operation, respectively. Except for the crossover operation, the remaining operations such as initial population, fitness functions, mutation, and reproduction, are the same as in the previous simulation. As we can see in this figure, the total execution time is reduced by an average of 3.08% with crossover operation compared to without crossover operation. We can see from the result of this figure that crossover operation can improve more performance rather than without crossover operation.

5.3 Effect of the Estimated Execution Analysis

In this simulation, we examined the performance of the estimated execution analysis according to the task size before and after GA. **Figs. 12**, **13**, **14**, and **15** show the total estimated execution time, estimated cost, estimated failure time, estimated rollback time, respectively. $Total_T$ is the sum of the estimated execution time in each instance, task distribution time, and task merge time. $Total_C$ denotes the sum of estimated costs for task execution in each instance. **Fig. 12** shows the total estimated execution time according to the task size before and after GA, respectively. After GA was applied, the total estimated execution time was reduced by an average of 33.59%. The total estimated execution times before and after GA were different because the instance failures occurred at different times. **Fig. 13** shows the estimated failure times before and after GA, respectively. The estimated failure time of all instances was reduced by an average of 15.45% after applying GA as compared to when GA was not applied. In **Fig. 14**, the estimated rollback time after GA showed an average performance improvement of 19.61% when compared to the rollback time before GA. The rollback time is calculated from a failure point to the last checkpoint time. Fig. 15 shows the estimated cost before and after applying GA, respectively. The cost after GA was decreased by an average of $0.12 as compared to the cost before GA. The difference between costs before and after applying GA is a little due to handling the equal task size. From the above results, we observe that the estimated execution times and failure times after applying GA were reduced as compared to before GA. Moreover, the costs were similar. The actual execution times and costs were compared based on above information. The improved results is had the reallocated task size according to GA. Because, our proposed GA is obtained relatively low the standard deviation about the estimated execution time than the standard deviation without GA.

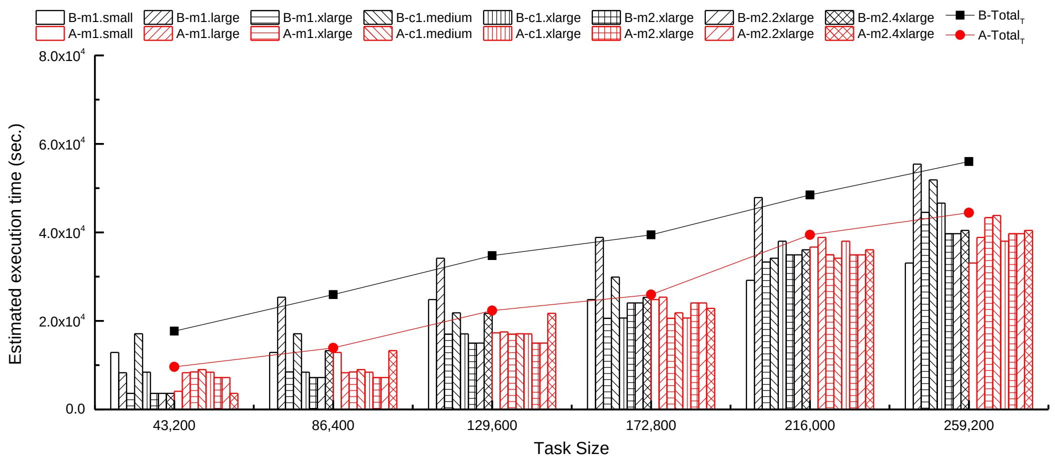


Fig. 12. Total estimated execution time

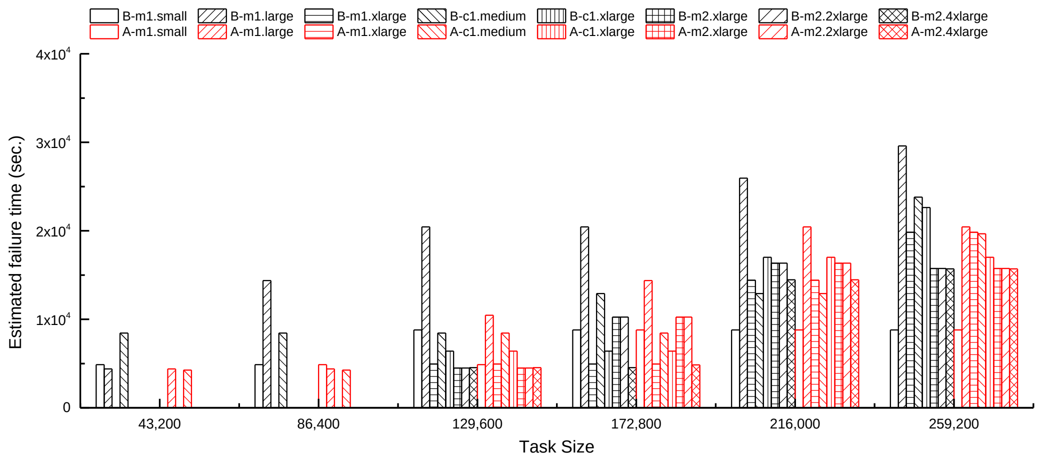


Fig. 13. Estimated failure time

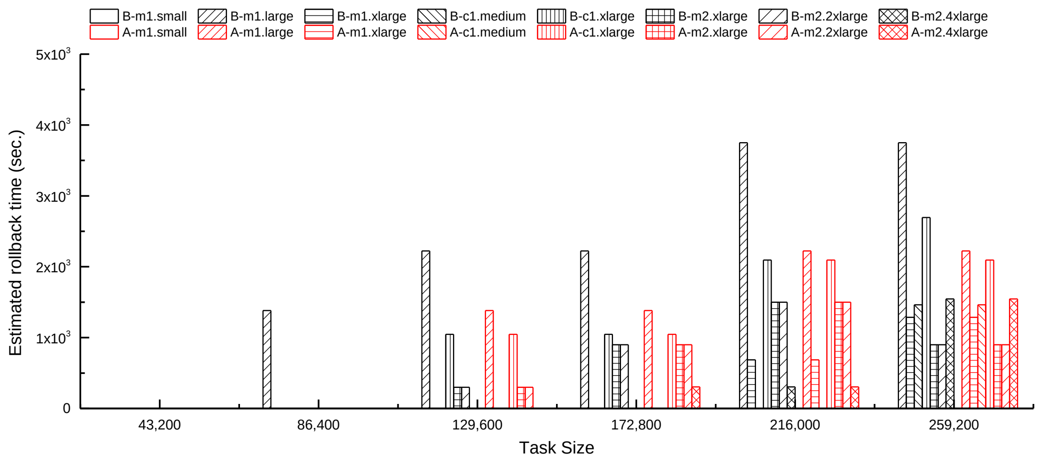


Fig. 14. Estimated rollback time

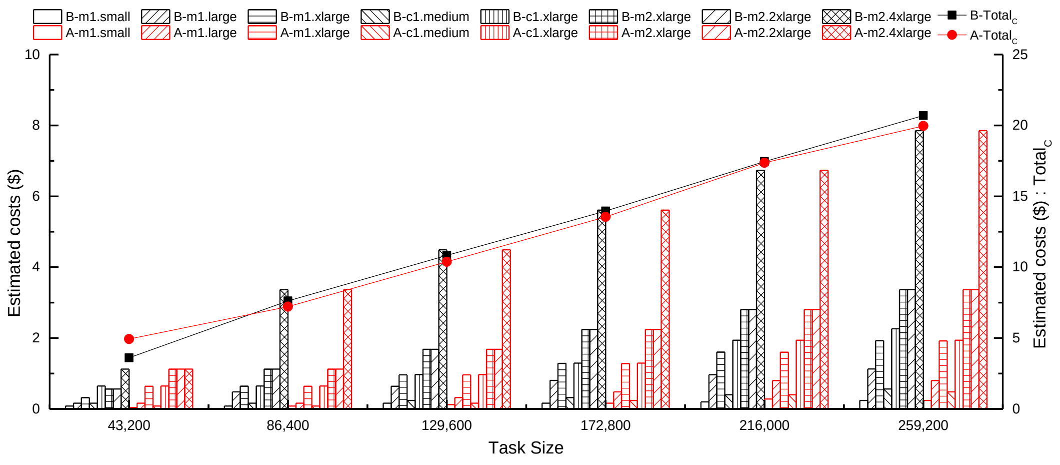


Fig. 15. Estimated costs

5.4 Effect of the Execution Analysis

In this simulation, we examined the performance of the execution analysis according to the task size before and after GA. **Figs. 16, 17, 18,** and **19** show the execution results of the actual data based on the estimated data, before and after GA. In the figures, $Total_T$ denotes the total time taken for the distribution and merging of tasks. $Total_C$ denotes the sum of costs of task execution in each instance. **Fig. 16** shows that the total execution time is reduced by an average of 7.06% after GA as compared to before GA. **Fig. 17** shows that the total costs after GA decreased the average by \$0.17 when compared to the cost before GA. **Fig. 18** shows that the failure time after GA was increased on average by 13.93% as compared to before GA. In **Fig. 19**, the rollback time after GA showed an average performance improvement of 6.52% when compared to the rollback time before GA. Different fluctuation of spot price contributes to difference between estimation and actual performances.

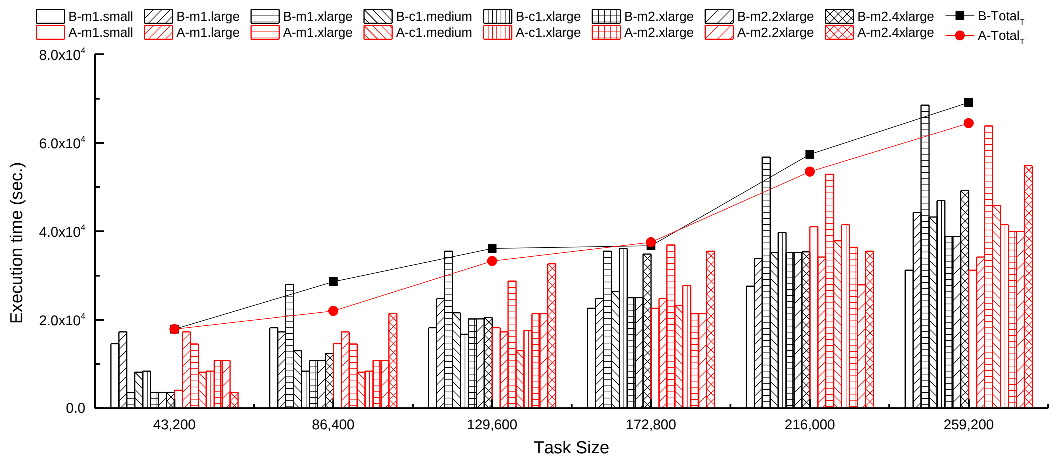


Fig. 16. Total execution time in task distribution

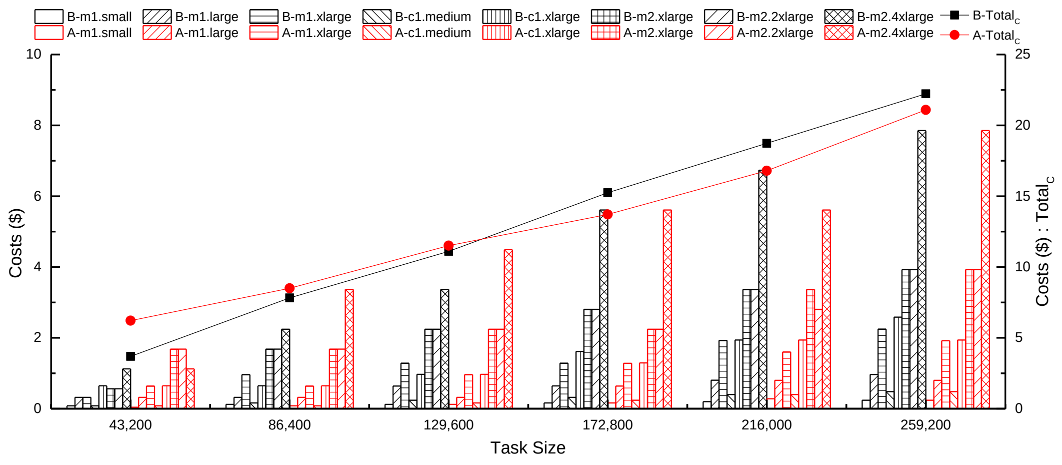


Fig. 17. Costs in task distribution

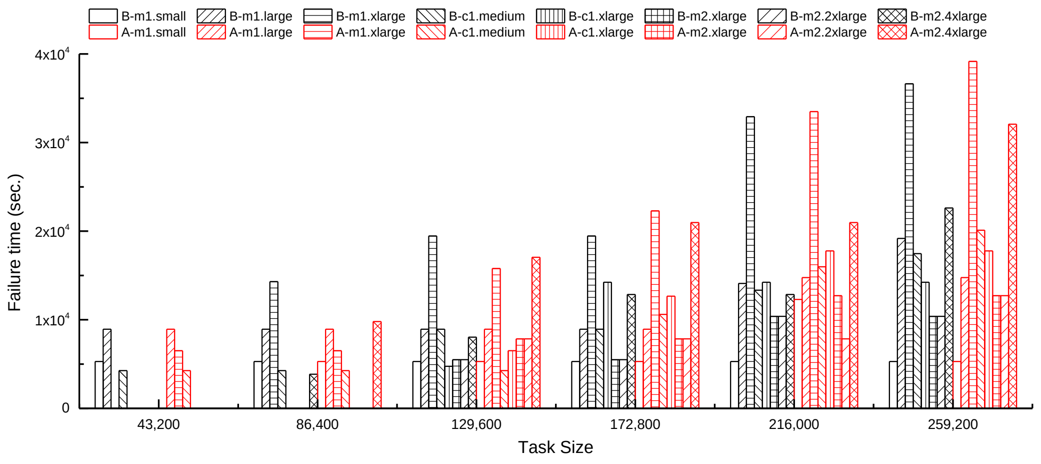


Fig. 18. Failure time in task distribution

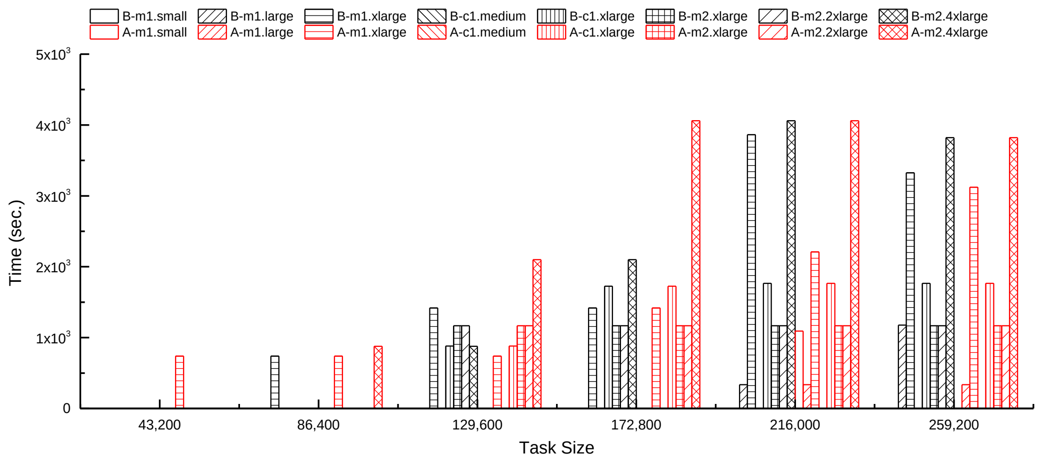


Fig. 19. Rollback time in task distribution

6. Conclusion

In this paper, we proposed a GA-based workflow scheduling technique for task distribution in unreliable cloud computing environments. In our environment, the resources can be unreliable anytime due to the fluctuation of instance prices, resulting in increasing the failure time of users' job. In order to solve the problem, we proposed GA-based workflow scheduling technique. The scheme proposed in this study reduced the failure time and the rollback time. The rollback time in our scheme was less than that of the existing scheme (without GA) because our scheme adaptively performs task distribution according to the estimated execution time of available instances. The simulation results showed that the execution time in our scheme was improved on average by 7.06% after GA as compared to before GA. Additionally, the failure time after applying GA was reduced on average by 6.52% as compared to before GA. Therefore, our scheduling method achieved minimizing the execution time and the cost of running applications. In future, we plan to expand our environment with an efficient GA operation that takes into consideration the current state of available instances.

7. Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2012R1A1A4A01015777).

References

[1] Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2, 2013.

[2] F.L. Ferraris, D. Franceschelli, M.P. Gioiosa, D. Lucia, D. Ardagna, E. Di Nitto, and T. Sharif, "Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds," in *Proc. of Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 423–429, 2012. Article (CrossRef Link).

[3] H.N. Van, F.D. Tran, and J.M. Menaud, "SLA-Aware Virtual Resource Management for Cloud Infrastructures," in *Proc. of Proceedings of the 2009 Ninth IEEE International Conference on*

Computer and Information Technology, vol. 2, pp. 357–362. IEEE Computer Society, 2009. Article (CrossRef Link).

[4] M. Komal, M. Ansuyia, and D. Deepak, "Round Robin with Server Affinity: A VM Load Balancing Algorithm for Cloud Based Infrastructure," *Journal of Information Processing Systems*, vol. 9, no. 3, pp. 379–394, 2013. Article (CrossRef Link).

[5] Hasan Sabbir and Eui-Nam Huh, "Heuristic based Energy-aware Resource Allocation by Dynamic Consolidation of Virtual Machines in Cloud Data Center," *KSII Transactions on Internet & Information Systems*, vol. 7, Issue 8, pp. 1825–1842, 2013. Article (CrossRef Link).

[6] Siqi Shen, Kefeng Deng, Alexandru Iosup, and Dick Epema, "Scheduling jobs in the cloud using on-demand and reserved instances," in *Proc. of Proceedings of the 19th international conference on Parallel Processing* (*Euro-Par'13*), pp. 242–254, 2013. Article (CrossRef Link).

[7] Amazon EC2 spot Instances, http://aws.amazon.com/ec2/spot-instances/, 2013.

[8] S. Yi, D. Kondo, and A. Andrzejak, "Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud," in *Proc. of Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, pp. 236–243. IEEE Computer Society, 2010. Article (CrossRef Link).

[9] G. Singer, I. Livenson, M. Dumas, S. N. Srirama, and U. Norbisrath, "Towards a model for cloud computing cost estimation with reserved resources," in *Proc. of Proceedings. of 2nd ICST International Conference on CloudComp 2010*, Barcelona, Spain. Springer, October 2010. Article (CrossRef Link).

[10] M. Mazzucco and M. Dumas, "Reserved or On-Demand Instances? A Revenue Maximization Model for Cloud Providers," in *Proc. of Proceedings of the 4th IEEE International CLOUD 2011*, pp. 428–435, July 2011. Article (CrossRef Link).

[11] S. Yi, J. Heo, Y. Cho, and J. Hong, "Taking point decision mechanism for page-level incremental checkpointing based on cost analysis of process execution time," *Journal of Information Science and Engineering*, vol. 23, no. 5, pp. 1325–1337, 2007. Article (CrossRef Link).

[12] William Voorsluys and Rajkumar Buyya., "Reliable Provisioning of Spot Instances for Compute-intensive Applications," in *Proc. of IEEE 26th International Conference on Advanced Information Networking and Applications*, 2012. Article (CrossRef Link).

[13] Qi Zhang, Eren Gurses, Raouf Boutaba, and Jin Xiao., "Dynamic resource allocation for spot markets in clouds," in *Proc. of the 11th USENIX conference Hot-ICE'11*, pp. 1–6, 2011. Article (CrossRef Link).

[14] Cloud exchange, http://cloudexchange.org, 2013.

[15] Goiri, F. Julia, J. Guitart, and J. Torres., "Checkpoint-based Fault-tolerant Infrastructure for Virtualized Service Providers," *12th IEEE/IFIP NOMS'10*, pp. 455–462, April 2010. Article (CrossRef Link).

[16] H. Fernandez, M. Obrovac, and C. Tedeschi, "Decentralised Multiple Workflow Scheduling via a Chemically-coordinated Shared Space," *INRIA Research Report*, RR-7925, pp. 1–14, 2012. Article (CrossRef Link).

[17] K. Liu, J. Chen, Y. Yang, and H. Jin, "A throughput maximization strategy for scheduling transaction-intensive workflows on SwinDeW-G," *Concurrency and Computation: Practice and Experience*, vol. 20, issue 15, pp. 1807–1820, 2008. Article (CrossRef Link).

[18] B. Hutt and K. Warwick, "Synapsing Variable-Length Crossover: Meaningful Crossover for Variable-Length Genomes," *IEEE Transactions on Evolutionary Computation*, vol. 11, issue 1, pp. 118 – 131, 2007. Article (CrossRef Link).

[19] John H. Holland, "Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence," *U. Michigan Press*, 1975. Article (CrossRef Link).

[20] Fullmer, Brad, and Risto Miikkulainen, "Using marker-based genetic encoding of neural networks to evolve finite-state behavior," in *Proc. of Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pp. 252–262, 1992. Article (CrossRef Link).

[21] J. Gu, J. Hu, Tianhai Zhao, and Guofei Sun, "A new resource scheduling strategy based on genetic

algorithm in cloud computing environment," *Journal of Computers*, vol. 7, no. 1, pp. 42–52, 2012. Article (CrossRef Link).

[22] S. Kaur and A. Verma, "An Efficient Approach to Genetic Algorithm for Task Scheduling in Cloud Computing Environment," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 4, no.10, pp. 74–79, 2012. Article (CrossRef Link).

[23] Fatma A. Omara and Mona M. Arafa, "Genetic algorithms for task scheduling problem," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, issue 7, pp. 758–766, 2010. Article (CrossRef Link).

[24] D. Jung, S. Chin, K. Chung, H. Yu, and J. Gil, "An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment," in *Proc. of Proceeding of NPC2011*, pp. 185–200, 2011. Article (CrossRef Link).

Daeyong Jung is Integrated Master and Ph.D. candidates in the Department of Computer Science Education of Korea University. He received his B.S. degree in Department of Electronic Engineering from Hanbat National University, Daejeon, Korea, in 2007, His main research interests are cloud computing, grid computing, distributed computing, and fault-tolerance systems.

Taeweon Suh received the Ph.D. degree in Electrical and Computer Engineering from the Georgia Institute of Technology. He is currently an associate professor in the Department of computer science and engineering at Korea University. His research interests include embedded systems, computer architecture, parallel computer architecture & programming, and computer science education.

Heonchang Yu received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University, Seoul, Korea, in 1989, 1991, and 1994, respectively. He has been a Professor of computer science and engineering with Korea University since 1998. From February 2011 to January 2012, he was a Visiting Professor of electrical and computer engineering in Virginia Tech. Since 2011, he has been the Director of Korea Information Processing Society, Korea. Prof. Yu is the Vice President of the Korean Association of Computer Education and an Editor of the Korean Institute of Information Technology. He was awarded the Okawa Foundation Research Grant of Japan in 2008. His research interests include cloud computing, grid computing, distributed computing, and fault-tolerant systems.

JoonMin Gil received the B.S. and M.S. degrees in computer science from Korea University, Korea, in 1994 and 1996, respectively, and the Ph.D. degree in computer science and engineering from Korea University, Korea in 2000. From 2001 to 2002, he was a Visiting Research Associate with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL, USA. From October 2002 to February 2006, he was a Senior Research Engineer with the Supercomputing Center, Korea Institute of Science and Technology Information (KISTI), Daejeon, Korea. In March 2006, he joined the Catholic University of Daegu, Korea, where he is currently an Associate Professor with the School of Information Technology Engineering. His recent research interests include cloud computing, distributed systems, wireless and sensor networks, and Internet computing.