A novel independent job rescheduling strategy for cloud resilience in the cloud environment

Fei Xie, Jun Yan and Jun Shen
School of Computing and Information Technology, University of Wollongong, Wollongong, Australia

Received 29 June 2021
Revised 3 January 2022
24 January 2022
Accepted 24 January 2022

Abstract

Purpose – Although proactive fault handling plans are widely spread, many unexpected data center outages still occurred. To rescue the jobs from faulty data centers, the authors propose a novel independent job rescheduling strategy for cloud resilience to reschedule the task from the faulty data center to other working-proper cloud data centers, by jointly considering job nature, timeline scenario and overall cloud performance.
Design/methodology/approach – A job parsing system and a priority assignment system are developed to identify the eligible time slots for the jobs and prioritize the jobs, respectively. A dynamic job rescheduling algorithm is proposed.
Findings – The simulation results show that our proposed approach has better cloud resiliency and load balancing performance than the HEFT series approaches.
Originality/value – This paper contributes to the cloud resilience by developing a novel job prioritizing, task rescheduling and timeline allocation method when facing faults.

Keywords Fault tolerance, Independent job rescheduling, Priority assignment system, Timeline allocation, Cloud resiliency, Load balancing performance

Paper type Research paper

1. Introduction

Although a cloud data center often has its own proactive fault handling plan, the threat of unplanned outages still exists [1–4]. Therefore, cloud resiliency is an important issue to successfully keep a stable cloud computing environment under fault scenarios [5]. To rescue a series of operations from a fault event, a variety of fault handling approaches have been proposed [6, 7]. Several parameters such as CPU temperature [6], cloud performance [8] and job importance [9], have been considered in these fault handling approaches. Job execution duration and job deadline, as two of the most significant job attributes, has also been considered when performing job scheduling, job rescheduling and resource allocation. Completing the job beyond its deadline is meaningless [10, 11]. From the point in time when a fault occurs, uncompleted jobs with the deadline requirements are particularly at risk of failing to meet their deadline requirements [12, 13].

Many deadline-constrained job scheduling strategies have been proposed. The HEFT series approaches are one of the most significant series of job scheduling strategies published from 2002 to date [14–23]. Although HEFT series approaches were proposed over past decade, selecting the first available server to enable job to finish early might not be the optimal solution when handling faults [15, 21, 24]. It may cause unnecessary deadline or resource competition between the job with high priority and the job with low priority. As a

Applied Computing and Informatics
Emerald Publishing Limited
e-ISSN: 2210-8327
p-ISSN: 2634-1964
DOI 10.1108/ACI-06-2021-0172

result, the cloud resiliency may not be optimized with many low priority jobs unsaved. Besides, selecting the first available server may cause a temporary dramatic load increase at some specific time points, which leads to performance bottleneck.

Therefore, in this paper, we propose a novel independent job rescheduling strategy for better cloud resiliency and load balancing performance. Our approach concentrates on independent job rescheduling based on job nature, timeline scenario and overall cloud performance. A job parsing system is deployed to identify the eligible time slots for the jobs. Then a priority assignment system is developed to prioritize the jobs in the cloud environment. To handle different cases, two sub-algorithms are applied in the proposed dynamic job rescheduling algorithm. The simulation results show that our proposed dynamic job rescheduling strategy has better cloud resiliency and load balancing performance than the HEFT series approaches. Besides, our approach can fit both the single-fault scenario and multi-faults scenario when handling faults.

2. Related work and problem statement

The cloud environment is subject to many types of faults, which might lead to a data center being unstable or even unavailable. These faults will disrupt uncompleted jobs [12]. Therefore, it is crucial for a data center to handle faults and rescue uncompleted jobs when faults occur [25, 26]. Fault tolerance can be approached from two different perspectives, proactive fault tolerance and reactive fault tolerance [27]. The main objective of fault tolerance techniques is to rescue the jobs from the faulty data center to working-proper replica-ready data centers [28, 29].

The deadline is one of the most significant parameter to be considered in some contemporary fault handling approaches. The HEFT series approaches are one of the most significant series of deadline-constrained job scheduling strategies, which are published from 2002 to date [14–23]. In 2002, a Heterogeneous Earliest Finish Time (HEFT) algorithm was proposed to minimize its earliest finish time with an inserted-based policy [14]. It firstly assigns the priority to each job in the scheduling list and then assigns each task to the first available server which can enable the job to finish the earliest. In 2013, a Budget and Deadline Constrained scheduling algorithm named BEFT was proposed to find the optimal workflow scheduling solution to satisfy both deadline and budget constraints for avoiding SLA violations [16]. Specifically, the BEFT algorithm only works by reserving and billing a fixed number of resources in heterogeneous grid computing systems. Same year in 2013, another novel list-based task scheduling algorithm called Predict Earliest Finish Time (PHEFT) was proposed to improve the makespan and efficiency to compare with the HEFT, LDCP and LHEFT approach. At the same time, this algorithm keeps the same time complexity to HEFT approach. In 2015, a Budget and Deadline Constrained Heterogeneous Earliest Finish Time (BDHEFT) algorithm was proposed by extending the classic HEFT approach and the BHEFT approach [18]. The BDHEFT approach considers six major variables, such as Spare Workflow Budget, Spare Workflow Deadline, Current Task Budget, Current Task Deadline, Budget Adjustment Factor and Deadline Adjustment Factor, to generate a Budget and Deadline Constrained scheduling plan. In 2016, an Enriched-Look ahead HEFT (E-LHEFT) algorithm was proposed to optimize both QoS and load balancing without considering any constraints [19]. E-LHEFT algorithm updates the processor selection phase of LHEFT algorithm by applying task grouping and Pareto theory for effective load balancing performance. In 2018, jobs with both unconstrained and time deadline constrained cases were taken into account by applying a HEFT technique for order preference called HEFT-T algorithm [20]. A three-stage non-dominated sorting approach is applied to identify the optimal solutions for the unconstrained case, and an adaptive weight adjustment strategy is proposed to adjust weight value for time for the deadline-constrained case. In 2019, a workflow scheduling algorithm

named Greedy Resource Provisioning and Modified HEFT (GRP-HEFT) was proposed by developing a resource provisioning mechanism [22]. The resource provisioning mechanism generates the instance type list based on the efficiency ratio of different instance types and selects the most efficient instances constrained by pre-defined budget. The modified HEFT algorithm employs the optimal configuration of instance types with their number of created VMs to obtain the job scheduling plan. Same year in 2019, a Dynamic Variant Rank HEFT (DVR-HEFT) algorithm was also proposed to reduce the scheduler's makespan without increasing the algorithm's time complicity to compare with the classic HEFT approach [23].

The HEFT series approaches tend to select the first available server to enable the earliest finish time. Although HEFT series approaches were developed over a long time period, selecting the first available server might not be the optimal configuration when handling faults [15, 21, 24]. It may cause anabatic deadline or resource competition problems in which the job rescue with the high priority may unnecessarily impact the job rescue with the low priority. Therefore, the cloud resiliency may be degraded. Besides, selecting the first available server may cause a temporary dramatic load increase at some specific time points. Therefore, the timeline processing should be further considered because of the time-varying characteristic of the entire timeline. Firstly, different time-varying resource situations at different cloud data centers should be taken into account. Secondly, the job deadline competition should be addressed when allocating the jobs. Furthermore, the load should be more balanced to avoid traffic spikes and degraded cloud performance.

3. General modeling

In general, a cloud environment contains multiple components such as data centers, servers, data and jobs. In this research, the proposed independent job rescheduling strategy is developed to globally control the job allocation over the cloud environment. We recognize a data center as an independent scheduling entity when doing job allocation. The job scheduling and rescheduling is conducted among a set of data centers. How the jobs are allocated into servers or virtual machines in a specific data center, is not the focus of this research. Instead, we study a cloud environment $E = (DC, J)$ with multiple data centers $DC\ \{dc_1, dc_2, \ldots, dc_p\}$ and a set of jobs $J\ \{j_1, j_2, \ldots, j_g\}$.

Each job $j \in J$ is associated with $R(j)$, DEAD(j) and PRO(j), which present resource requirement, job deadline and job operation profit, respectively. Each job j has a fixed job execution duration Len(j). The job urgency value of the job j, $\rho(j)$, refers to the time buffer between current time point and its deadline DEAD(j). In this research, we define the job deadline as a specific point in time without the consideration of the job with infinite deadline. Completing a job beyond its deadline is meaningless. All jobs in this research are independent, which means there is no dependency among the jobs. Besides, we assume all jobs require restarting of the entire job as the traditional job rescheduling approach did [30]. This is because the major concern of this paper is not the way how the job is resumed. The main objective is on how to maximize the overall cloud resiliency and balance the resource load by analyzing the job priority. The traditional restart approach is also the most general approach that is supported by any cloud for any task.

In this research, we focus on the improvement of the cloud resiliency. The cloud resiliency for a faulty data center can be calculated using the formula in Eqn (1).

$$\text{Cloud Resiliency} = \frac{\text{Total number of rescued jobs}}{\text{Total number of jobs to be rescued}} \tag{1}$$

4. Job parsing system

We develop a job parsing system to find the eligible time slots for jobs. A timeline exists at each data center. In this research, we do not consider the job with an infinite deadline. Therefore, the timeline range refers to $[T_0,\ T_{\text{Latest}}]$, where T_0 denotes the fault-occurred time point and T_{Latest} denotes the latest deadline time point of the jobs in J.

We define the time slot as a series of continuous time points. The available resource at each time point is the most significant factor for the further reception of the rescheduled jobs from the faulty data center. Therefore, we parse the timeline at each data center site in a two dimensional vector space. The x axis is the discrete time points ranged from $[T_0,\ T_{\text{Latest}}]$ and the y axis is the available resource. Thus, the line in this space represents the available resource over time. We call it a resource line. Each job can be parsed into this two dimensional vector space as a rectangle. The height of the rectangle represents the resource requirement of the job and the length of the rectangle corresponds to the job execution duration. The rectangle will horizontally move from T_0 to T_{Latest}. An eligible time slot for a job starts from a time point when the rectangle starts to stand completely below the resource line and ends at a time point when the rectangle starts to stand above the resource line. A function $\text{Count}(S(j)^{dc})$ is deployed to count the number of eligible time slots of the job j at a data center dc. An example of our proposed job parsing system is shown in Figure 1. The final range of the eligible time slot can be recognized when the job rectangle completely stands under the resource line (red line).

5. Job rescheduling strategy

Our job scheduling strategy has three phases, replica selection phase, job prioritizing phase and eligible time slot selection phase. Our strategy is an independent job rescheduling strategy for a bounded number of data centers when faults occur. In the case of single-fault scenario, our strategy can be applied by the faulty data center in one time to rescue the faulty jobs. While in the case of multi-faults scenarios, our strategy should be separately applied in each faulty data center. The main objective of our job rescheduling strategy is to maximize the cloud resiliency and balance the resource load at the same time by applying a priority-based job rescheduling method. Therefore, the starvation of the less privileged jobs may be sacrificed to some extent as our proposed job prioritizing method aims to obtain the job importance by analyzing the job urgency, the job operation profit and the number of eligible

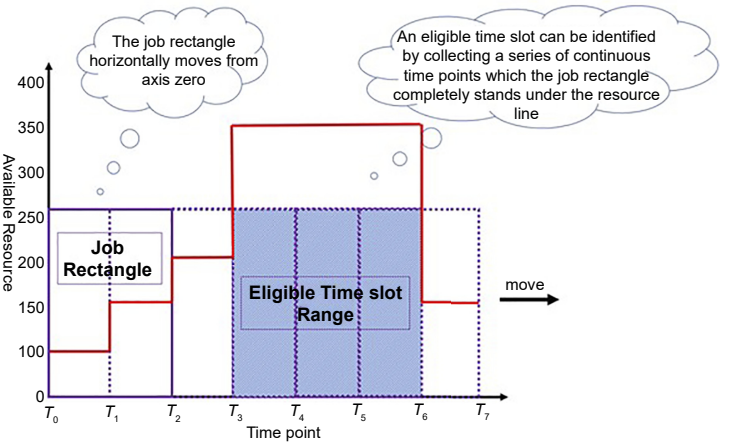

Figure 1.
The example of eligible time slot identification

time slots of the job. The job urgency describes the urgency degree of the job. The job operation profit demonstrates the economic value of the job completion. The number of eligible time slots of the job denotes the job allocation difficulty. Therefore, the job with higher priority means the job has higher urgency degree, higher economic value and higher job allocation difficulty. In other words, the job with lower priority means the job has lower urgency degree, lower economic value and lower job allocation difficulty. Therefore, from the relation among urgency degree, economic value and job allocation difficulty, the less privileged jobs are less important to be saved. Therefore, the resource will be firstly available to the jobs with higher priority and then the starvation of the jobs is sacrificed to some extent.

5.1 Replica selection phase

Our approach has a performance-oriented replica selection policy which selects the replica-ready data center which has the most available resource as the optimal rescheduling destination. Tie-breaking is done randomly.

5.2 Job prioritizing phase

This phase distributes the job allocation priority to each job. The job allocation priority list will preserve a descending processing order based on the job priority until no more jobs can be allocated. Tie-breaking is done randomly.

We develop a priority assignment system to assign the job allocation priority by jointly taking job urgency, job operation profit and the number of eligible time slots of the job into account. Each job can be parsed to a cuboid in a three dimensional vector space, where the cuboid length a denotes the job urgency value $\rho(j)$ on the y axis, the cuboid height c represents the reciprocal of the job operation profit, $\frac{1}{\mathrm{PRO}(j)}$, on the z axis and the cuboid width b denotes the number of eligible time slots of the job j on the x axis. According to the parsing method above, the volumes among cuboids will be compared. The smaller volume the cuboid has, the more urgent, the more profitable and the more processing difficulty the job has. Hence, the cuboid with the smaller volume has higher priority. The job allocation priority list is created based on the volume value of each job cuboid.

5.3 Eligible time slot selection phase

Our eligible time slot selection method aims to select the optimal eligible time slot for the received jobs at each working-proper data center. Scenario-based allocation are applied for both normal cases (Algorithm 2) and limited resource or insufficient time slot length cases (Algorithm 3). Tie-breaking is done randomly.

We consider both the time slot length similarity and the corresponding time slot resource situations to accommodate the job at its optimal eligible time slot. The eligible time slot with the more similar time slot length similarity to the job execution duration is more suitable to accommodate the job for less the time slot space wastefulness. The higher minimum available resource in the eligible time slot achieves the less possibility of load spike.

Let $S(j)^{dc}$ denotes a set of eligible time slots of the job j at a data center dc, then $s(j)_q^{dc}$ denotes the q th eligible time slot in $S(j)^{dc}$. $\delta(s(j)_q^{dc})$ denotes the time slot length similarity of $s(j)_q^{dc}$ to the rescheduled job execution duration and $\sigma(s(j)_q^{dc})$ denotes the minimum available resource value of $s(j)_q^{dc}$. Then the credit of the $s(j)_q^{dc}$, $C(s(j)_q^{dc})$, is formulated as follows in Eqn (2). W_δ is the weight of the ranking value of $\delta(s(j)_q^{dc})$ and W_σ is the weight of the ranking value of $\sigma(s(j)_q^{dc})$. The sum of W_δ and W_σ is 1. $\mathrm{rank}(\delta(s(j)_q^{dc}))$ denotes the ranking value of the time slot length similarity of $s(j)_q^{dc}$ and $\mathrm{rank}(\sigma(s(j)_q^{dc}))$ denotes the ranking value of the minimum resource of $s(j)_q^{dc}$.

$$\begin{cases} \delta\left(s(j)_q^{dc}\right) = \mathrm{Len}\left(s(j)_q^{dc}\right) - \mathrm{Len}(j) \\ \mathrm{rank}\left(\delta\left(s(j)_q^{dc}\right)\right) = \dfrac{\max\left(\delta\left(S(j)^{dc}\right)\right) - \delta\left(s(j)_q^{dc}\right)}{\max\left(\delta\left(S(j)^{dc}\right)\right) - \min\left(\delta\left(S(j)^{dc}\right)\right)} \\ \mathrm{rank}\left(\sigma\left(s(j)_q^{dc}\right)\right) = \dfrac{\sigma\left(s(j)_q^{dc}\right) - \min\left(\sigma\left(S(j)^{dc}\right)\right)}{\max\left(\sigma\left(S(j)^{dc}\right)\right) - \min\left(\sigma\left(S(j)^{dc}\right)\right)} \\ C\left(s(j)_q^{dc}\right) = W_\delta * \mathrm{rank}\left(\delta\left(s(j)_q^{dc}\right)\right) + W_\sigma * \mathrm{rank}\left(\sigma\left(s(j)_q^{dc}\right)\right) \end{cases} \tag{2}$$

Then the eligible time slot of the job j with the maximum credit will be recognized as the optimal eligible time slot $O(j)$ for the job j.

To implement three phases mentioned above, a dynamic job rescheduling algorithm is proposed in Algorithm 1. The time complexity of Algorithm 1 is $O(n^3)$. The algorithm firstly initializes job parsing vector space and makes the job collection at the faulty data center from *Line 1* to *Line 4*. Then we select the optimal replica-ready data center for each job at the faulty data center from *Line 5* to *Line 7*. We prioritize the received jobs at each working-proper data center site from *Line 10* to *Line 12*. Each data center will try to find an eligible time slot for their received jobs from *Line 13* to *Line 25*. Two different scenarios can be treated during this process. Algorithm 2 is normally applied between *Line 13* to *Line 18* if the eligible time slots can be founded. Otherwise, Algorithm 3 is applied between *Line 19* and *Line 23* for the insufficient time slot length cases.

Algorithm 1: Job Rescheduling Algorithm

1. Initialization {
2. Set timeline
3. Create job parsing vector spaces
4. Load the jobs at the faulty data center into faulty job list $FJ[]$ }
5. Remove the faulty data center from DC
6. **for** each job in $FJ[]$ **do**
7. Select the optimal replica-ready data center
8. **end for**
9. **for** each dc in DC **do**
10. Parse the jobs and the timeline
11. Prioritize the jobs
12. Create priority list $AP[]$ in an ascending order

13. **for** $AP[0]$ to $AP[Sizeof(AP[]) - 1]$ **do**
14. $Count(S(AP[])^{dc})$
15. **if** $Count(S(AP[])^{dc}) > 0$
16. Do Algorithm 2
17. Move $AP[] \rightarrow O(AP[])$
18. **else**
19. Do Algorithm 3
20. Load "Optimal Migratable Job" and "Alternative Migration Destination"
21. Migrate "Optimal Migratable Job" $\rightarrow T_{Begin}$ of "Alternative Migration Destination"
22. Record the original location of "Optimal Migratable Job" as $O(AP[])$
23. Move $AP[] \rightarrow O(AP[])$
24. **end if**
25. **end for**
26. **end for**

Algorithm 2 is used to generate the optimal eligible time slot $O(j)$ for job j, which assists our proposed Algorithm 1. The time complexity of Algorithm 2 is $O(1)$. In our Algorithm 2, we firstly insert the job from $AP[]$ at *Line 1*. Then we find the optimal eligible time slot for the inserted job from *Line 2* to *Line 7*. The credits for the eligible time slots will be calculated for the inserted job at *Line 2* under Eqn (2). After that, the optimal eligible time slot for the inserted job will be generated at *Line 3*. The optimal eligible time slot will be loaded to find its beginning time point T_{Begin} at *Line 4*[]. The inserted job should be allocated to the beginning time point T_{Begin} of the optimal eligible time slot at *Line 5*. The resource consumption of the inserted job should be updated to resource line in the vector space at *Line 6*. Finally, the priority assignment system will be updated for further cloud resilience at *Line 7*. In our approach, we commonly allocate the rescheduled job at the first time point (the beginning time point) in the optimal eligible time slot because we insist "as early as possible" principle for all job completeness. For tie-breaking eligible time slots, we place the job at the earliest available time slot as well.

Algorithm 2: Optimal Eligible Time Slot Selection

Input: $AP[k]$, $k \in \{0,1, \ldots, Sizeof(AP[]) - 1\}$

Output: $O(AP[k])$

1. Insert the job $AP[k]$ at Line 16 in Algorithm 1
2. Calculate $C(S(AP[k])^{dc})$ under Eq. 2
3. Generate $O(AP[k])$
4. Load T_{Begin} of $O(AP[k])$
5. Allocate $AP[k]$ at T_{Begin}
6. Update resource line for $O(AP[k])$
7. Update priority assignment system and generate new allocation priority list

By applying Algorithm 2, our approach can rescue the jobs that already has eligible time slots. The jobs that are left unsaved are known as residual jobs because of unsuccessful rescue due to insufficient resource or insufficient number of eligible time slots. We apply a residual job allocation in Algorithm 3 by using job migration methods. The time complexity of Algorithm 3 is $O(n)$.

In Algorithm 3, we firstly transform $FJ[]$ to "Residual Job List" $RJ[]$ and then load current running jobs in the environment into a new job list called "Current Running List" $CR[]$ from *Line 1* to *Line 2*. Then we try to find the probable eligible time slots for each residual job by following the order of $RJ[]$ from *Line 3* to *Line 15*. If the bottom of $RJ[]$ is reached, then we will end the Algorithm 3 at *Line 65*. Otherwise, the current rest time slots which meets $R(RJ[i])$ will be identified and then add into "Pro bable Eligible Time Slot List" $PRE[]$ from *Line 7* to *Line 8*.

If the current rest time slots which meets $R(RJ[i])$ cannot be founded, then the next residual job will be processed at *Line 10*. Otherwise, the probable released jobs will be founded from *Line 16* to *Line 40*. We firstly compare the resource requirement between $RJ[i]$ and $CR[m]$ at *Line 17*. According to the resource requirement comparison, the probable released job list $PRJ[]$ can be identified from *Line 18* to *Line 23*.

Then the capacity of $PRJ[]$ will be checked. If $PRJ[]$ is empty, the next residual job will be processed at *Line 25*. Otherwise, the probable released job list $PRJ[]$ will be filtered from *Line 27* to *Line 38*. The current running jobs will be removed from $PRJ[]$ if they are discrete to $PRE[]$ from *Line 28* to *Line 29*.

After the remove operations, the capacity of $PRJ[]$ will be checked once again. If $PRJ[]$ is empty, the next residual job will be processed at *Line 32*. Otherwise, a new job list called "Ready-to-Release List" $RTR[]$ will be created at *Line 37*. Then we try to find an alternative eligible time slot for the jobs in $RTR[]$ to continuously ensure the released job completeness from *Line 41* to *Line 50*. By filtering the $RTR[]$, a new job list called "Migratable List" $MIG[]$ will be created at *Line 46*.

Then we try to determine the released job from $MIG[]$ from *Line 51* to *Line 64*. The "Migratable List" $MIG[]$ will be firstly ordered based on job execution duration in an ascending order at *Line 51*. By following the order in $MIG[]$, an alternative eligible time slot will be tried to identify for the migratable job in $MIG[]$ *at Line 53*. If the alternative eligible time slot can be founded, the migratable job in $MIG[]$ will be labeled as "Optimal Migratable Job" and the alternative eligible time slot will be labeled as "Alternative Migration Destination". Otherwise, the next job in $MIG[]$ will be tried until "Optimal Migratable Job" is founded. If "Optimal Migratable Job" cannot be founded, then the next residual job will be processed at *Line 61*.

Algorithm 3: Residual Job Allocation

Input: $FJ[]$, dc

Output: Residue job allocation solution

1. Transform $FJ[]$ to residual job list $RJ[]$
2. Load current running jobs into current running list $CR[]$
3. **for** each $RJ[i]$ in $RJ[]$, $i = 0$, $i \leq Sizeof(RJ[]) - 1$] **do**
4. **if** the bottom of $RJ[]$ is reached
5. Go to Line 65
6. **else**
7. Identify current rest time slots which meets $R(RJ[i])$
8. Add into probable eligible time slot list $PRE[]$
9. **if** $PRE[] = NULL$
10. Back to Line 3 and $i++$

11. **else**
12. Go to Line 16
13. **end if**
14. **end if**
15. **end for**
16. **for** each $CR[m]$ in $CR[]$, $m = 0$, $m \leq Sizeof(CR[]) - 1]$ **do**
17. Compare $R(RJ[i])$ and $R(CR[m])$
18. **if** $R(CR[m]) > R(RJ[i])$
19. Add $CR[m]$ into probable released job list $PRJ[]$
20. Go to Line 24
21. **else**
22. $m++$
23. **end if**
24. **if** $PRJ[] = NULL$
25. Back to Line 3 and $i++$
26. **else**
27. Filter $PRJ[]$
28. **if** $CR[m]$ is discrete to $PRE[]$
29. Remove $CR[m]$ from $PRJ[]$
30. Check $PRJ[]$
31. **if** $PRJ[] = NULL$
32. Back to Line 3 and $i++$
33. **else**
34. Go to Line 37
35. **end if**
36. **else**
37. Add $CR[m]$ into ready-to-release list $RTR[]$
38. **end if**
39. **end if**
40. **end for**
41. **for** each $RTR[n]$ in $RTR[]$, $n = 0$, $n \leq Sizeof(RTR[]) - 1]$ **do**
42. Try to release $RTR[n]$
43. Evaluate after-release time slot length
44. Test the feasibility of after-release time slot length to accommodate $RJ[i]$
45. **if** Line 44 = $TRUE$
46. Add $RTR[n]$ into migratable list $MIG[]$

47. **else**
48. Back to Line 3 and $i++$
49. **end if**
50. **end for**
51. Order $MIG[]$ based on job execution duration in an ascending order
52. **for** $MIG[0]$ to $MIG[Sizeof(MIG[])-1]$ **do**
53. Find an alternative eligible time slot
54. **If** Line 53 = $TRUE$
55. Label $MIG[]$ as "Optimal Migratable Job"
56. Label the alternative eligible time slot as "Alternative Migration Destination"
57. Back to Line 3 and $i++$
58. **else**
59. Move to the next job in $MIG[]$
60. **if** the bottom of $MIG[]$ is reached
61. Back to Line 3 and $i++$
62. **end if**
63. **end if**
64. **end for**
65. End Algorithm 3

6. Simulation results

To evaluate our proposed approach, we performed three simulations on OMNeT++ 5.4.1. We make the following assumptions in our simulations:

(1) Traditional three-replicas strategy is deployed.

(2) The latency among data centers is insignificant.

(3) All data centers are inter-connected.

(4) Bandwidth is set as consumed resource.

(5) W_δ and W_σ is set to be 0.5 each to simplify the problem.

We implemented three types of workflows, Montage scientific workflow, LIGO Inspiral Analysis workflow and SIPHT program. Each workflow is seen as an independent job instance. The details of these workflows are adjusted and referenced from [31]. We measure the cloud resiliency in all three simulations and the load situations specifically in Simulation 2 to compare with the job scheduling method of the HEFT series approaches. We compare the performance of our approach to the average performance of HEFT series approaches.

6.1 Simulation 1 – multiple types of jobs with different deadlines

A cloud environment of 4 data centers with 6 circuits of 100 Gbps optical-fiber network integrated at each data center is set up in Simulation 1. The fault occurs at T_0 in data center dc_1.

In Simulation 1, the job input rule is set as follows. We input 200 jobs per input round.

(1) We input a random number of two types of jobs out of total 200 jobs per input round when resource is sufficient.

(2) We only input feasible input combinations per input round to the environment when resource is insufficient.

The simulation result is shown in Figure 2. It is obvious that our proposed approach has better cloud resiliency than the HEFT series approaches. As the number of jobs increased from 400 to 1400, our approach continued to rescue 100% of the faulty jobs. The HEFT series failed to rescue 100% faulty jobs when the number of jobs exceeds 600. The cloud resiliency of our approach dropped to 74.67% at 1600 jobs due to resource limitations and insufficient eligible time slots. However, our approach still keeps greater cloud resiliency than that of HEFT series approaches at 1600 jobs.

6.2 Simulation 2 – expanded cloud scale and load testing

In Simulation 2, we not only evaluate our cloud resiliency but also our load balancing performance from expanded cloud scale. The better load balancing performance helps cloud service providers avoid the traffic spikes and degraded performance. A cloud environment of 4 data centers with 60 circuits of 100 Gbps optical-fiber network integrated at each data center was developed. The fault occurs at T_0 in the data center dc_1.

In Simulation 2, the job input rule is set as follows. We input 1000 jobs per input round.

(1) We input a random number of random types of jobs out of total 1000 jobs per input round when resource is sufficient.

(2) We only input feasible input combinations per input round to the environment when resource is insufficient.

The simulation result is shown in Figure 2. It is also obvious that our proposed approach still has better cloud resiliency than the HEFT series approaches when the cloud scale expands. As the number of jobs increased from 9000 to 14000, our approach continued to keep 100% cloud resiliency. The HEFT series approaches fail to rescue all jobs when the number of jobs exceeds 9000. The cloud resiliency of our approach dropped to 51.03% at 15000 jobs because of the same reason in Simulation 1. However, our approach still keeps higher cloud resiliency than that of HEFT series approaches at 15000 jobs.

In this simulation, we also test the load situations at each time point for all working-proper data centers. The resource load situations are shown in Figure 3. The HEFT series approaches remain a peak load between T_0 and T_{2500} in dc_2 and dc_4, and then has a sharp load decrease. It leaves a long-time idle load after T_{2500} in dc_2 and dc_4 and makes a crowd load before that time point. However, our approach significantly reduces the load before T_{2500} in dc_2 and dc_4, and balances the load to the suitable time points at all three working-proper data centers. Although we still have some short-time peak load, our approach is obviously better than the HEFT series approaches in terms of load balancing. It means we achieve more balanced load to avoid load spikes.

7. Conclusions and future work

To conclude, the HEFT series approach is one of the most significant deadline-constrained job scheduling approaches. But selecting the first available server might not the optimal configuration when handling faults. In this paper, we propose a novel job rescheduling strategy for better cloud resiliency and Load balancing performance. This approach concentrates on independent job rescheduling based on job nature, timeline scenario and overall cloud performance to handle the job rescue from the faulty data center. A job parsing

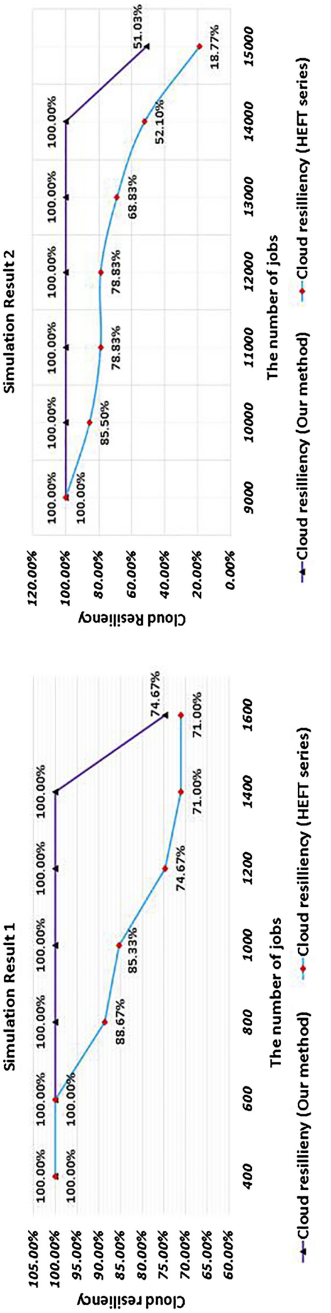

Figure 2.
Simulation result 1 and 2

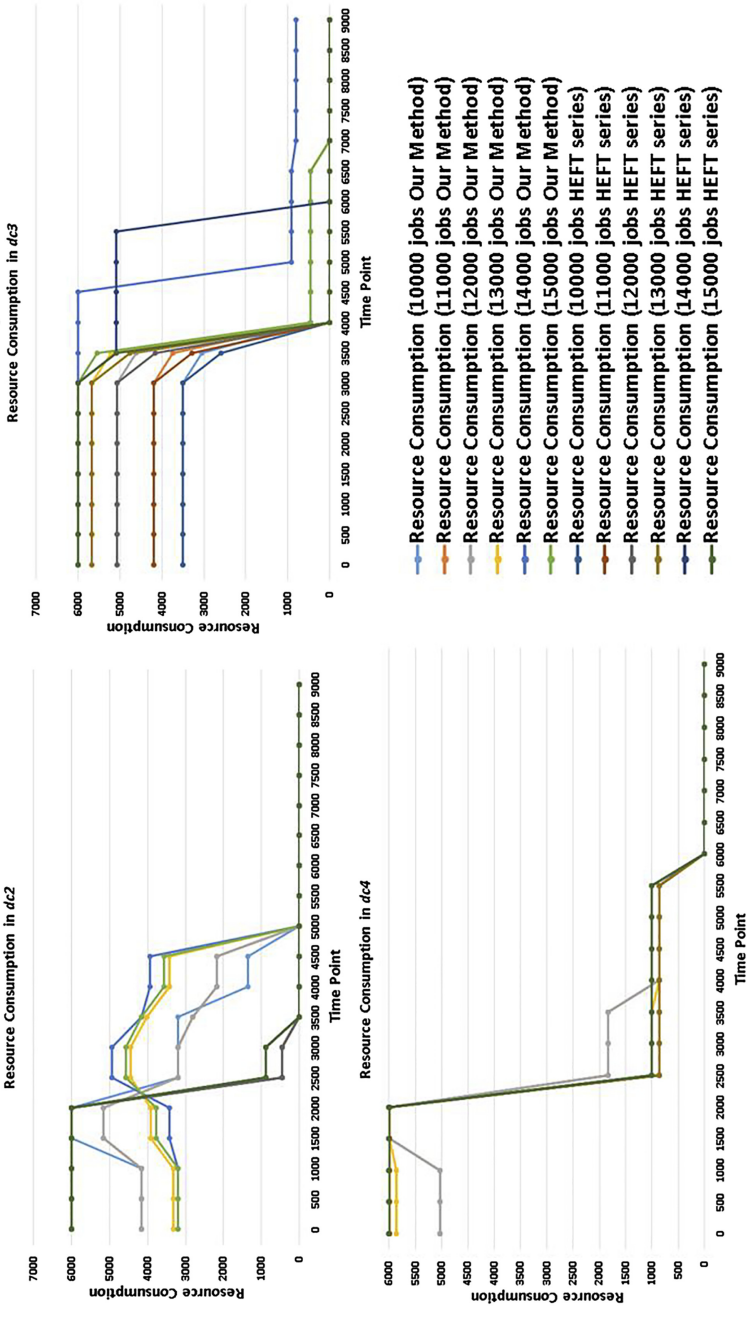

Figure 3.
Resource load in different data centers

system and a priority assignment system are developed to identify the eligible time slots for the jobs and prioritize the jobs, respectively. Two algorithms (Algorithm 2 and 3) are proposed to support the proposed dynamic job rescheduling algorithm (Algorithm 1). The simulation results show that our proposed strategy has better cloud resiliency and load balancing performance than the HEFT series approaches. Besides, both single-fault scenario and multi-faults scenario can adopt our strategy. In the future work, we propose to further develop a fault handling approach for dependent jobs. Apart from that, the check pointing method will be considered instead of the restart method for tasks in this approach.

References

1. Sampaio AM, Barbosa JG. A comparative cost analysis of fault-tolerance mechanisms for availability on the cloud. Sustai Comput Inform Syst. 2018; 19: 315-23. doi: 10.1016/j.suscom.2017.11.006.
2. Sivagami VM, Easwarakumar KS. An improved dynamic fault tolerant management algorithm during VM migration in cloud data center. Future Generat Comput Syst. 2019; 98: 35-43. doi: 10.1016/j.future.2018.11.002.
3. Ray B, Saha A, Khatua S, Roy S. Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment. IEEE Trans Cloud Comput. 2020. doi: 10.1109/TCC.2020.2968522.
4. Tomás L, Kokkinos P, Anagnostopoulos V, Feder O, Kyriazis D, Meth K, Varvarigos E, Varvarigou T. Disaster recovery layer for distributed OpenStack deployments. IEEE Trans Cloud Comput. 2017; 8(1): 112-23. doi: 10.1109/TCC.2017.2745560.
5. Shetty J, Babu BS, Shobha G. Proactive cloud service assurance framework for fault remediation in cloud environment. Int J Electr Computer Eng. 2020; 10(1): 987-96. doi: 10.11591/ijece.v10i1.pp987-996.
6. Liu J, Wang S, Zhou A, Kumar S, Yang F, Buyya R. Using proactive fault-tolerance approach to enhance cloud service reliability. IEEE Trans Cloud Comput. 2016; 6(4): 1191-202. doi: 10.1109/TCC.2016.2567392.
7. Zhou A, Wang S, Cheng B, Zheng Z, Yang F, Chang RN, Lyu MR, Buyya R. Cloud service reliability enhancement via virtual machine placement optimization. IEEE Trans Serv Comput. 2017; 10(6): 902-13. doi: 10.1109/TSC.2016.2519898.
8. Deng S, Huang L, Taheri J, Zomaya AY. Computation offloading for service workflow in mobile cloud computing. IEEE Trans Parallel Distributed Syst. 2014; 26(12): 3317-29. doi: 10.1109/TPDS.2014.2381640.
9. Vardhan M, Goel A, Verma A, Kushwaha DS. A dynamic fault tolerant threshold based replication mechanism in distributed environment. Proc Technol. 2012; 6: 188-95. doi: 10.1016/j.protcy.2012.10.023.
10. Zhu X, Wang J, Guo H, Zhu D, Yang LT, Liu L. Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds. IEEE Trans Parallel Distributed Syst. 2016; 27(12): 3501-17. doi: 10.1109/TPDS.2016.2543731.
11. Marahatta A, Wang Y, Zhang F, Sangaiah AK, Tyagi SKS, Liu Z. Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers. Mobile Netw Appl. 2019; 24(3): 1063-77. doi: 10.1007/s11036-018-1062-7.
12. Poola D, Garg SK, Buyya R, Yang Y, Ramamohanarao K. Robust scheduling of scientific workflows with deadline and budget constraints in clouds. In: IEEE 28th International Conference on Advanced Information Networking and Applications; 2014. p. 858-65. doi: 10.1109/AINA.2014.105.
13. Yao G, Ding Y, Hao K. Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems. IEEE Trans Parallel Distributed Syst. 2017; 28(12): 3671-83. doi: 10.1109/TPDS.2017.2687923.
14. Topcuoglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distributed Syst. 2002; 13(3): 260-74. doi: 10.1109/71.993206.

15. Bittencourt LF, Sakellariou R, Madeira ER. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In: 18th Euromicro Conference on Parallel. Distributed and Network-based Processing; 2010. p. 27-34. doi: 10.1109/PDP.2010.56.

16. Zheng W, Sakellariou R. Budget-deadline constrained workflow planning for admission control. J Grid Comput. 2013; 11(4): 633-51. doi: 10.1007/s10723-013-9257-4.

17. Arabnejad H, Barbosa JG. List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans Parallel Distributed Syst. 2013; 25(3): 682-94. doi: 10.1109/TPDS.2013.57.

18. Verma A, Kaushal S. Cost-time efficient scheduling plan for executing workflows in the cloud. J Grid Comput. 2015; 13(4): 495-506. doi: 10.1007/s10723-015-9344-9.

19. Shakkeera L, Tamilselvan L. QoS and load balancing aware task scheduling framework for mobile cloud computing environment. Int J Wireless Mobile Comput. 2016; 10(4): 309-16. doi: 10.1504/IJWMC.2016.078201.

20. Liu L, Fan Q, Buyya R. A deadline-constrained multi-objective task scheduling algorithm in mobile cloud environments. IEEE Access. 2018; 6: 52982-96. doi: 10.1109/ACCESS.2018.2870915.

21. Samadi Y, Zbakh M, Tadonki C. E-HEFT: enhancement heterogeneous earliest finish time algorithm for task scheduling based on load balancing in cloud computing. In: International Conference on High Performance Computing & Simulation (HPCS); 2018. p. 601-9. doi: 10.1109/HPCS.2018.00100.

22. Faragardi HR, Sedghpour MRS, Fazliahmadi S, Fahringer T, Rasouli N. GRP-HEFT: a budget-constrained resource provisioning scheme for workflow scheduling in IaaS clouds. IEEE Trans Parallel Distributed Syst. 2019; 31(6): 1239-54. doi: 10.1109/TPDS.2019.2961098.

23. Sandokji S, Eassa F. Dynamic variant rank HEFT task scheduling algorithm toward exascle computing. Proc Comp Sci. 2019; 163: 482-93. doi: 10.1016/j.procs.2019.12.131.

24. Setlur AR, Nirmala SJ, Singh HS, Khoriya S. An efficient fault tolerant workflow scheduling approach using replication heuristics and checkpointing in the cloud. J Parallel Distributed Comput. 2020; 136: 14-28. doi: 10.1016/j.jpdc.2019.09.004.

25. Cheraghlou MN, Khadem-Zadeh A, Haghparast M. A survey of fault tolerance architecture in cloud computing. J Netw Computer Appl. 2016; 61: 81-92. doi: 10.1016/j.jnca.2015.10.004.

26. Yu CY, Lee CR, Tsao PJ, Lin YS, Chiueh TC. Efficient group fault tolerance for multi-tier services in cloud environments. In: IEEE International Conference on Communications (ICC); 2020; p. 1-7. doi: 10.1109/ICC40277.2020.9149253.

27. Mukwevho MA, Celik T. Toward a smart cloud: a review of fault-tolerance methods in cloud systems. IEEE Trans Serv Comput. 2018. doi: 10.1109/TSC.2018.2816644.

28. Hasan M, Goraya MS. Fault tolerance in cloud computing environment: a systematic survey. Comput Industry. 2018; 99: 156-72. doi: 10.1016/j.compind.2018.03.027.

29. Ragmani A, Elomri A, Abghour N, Moussaid K, Rida M, Badidi E. Adaptive fault-tolerant model for improving cloud computing performance using artificial neural network. In: Procedia computer science. 2020: 170: 929-34. doi: 10.1016/j.procs.2020.03.106.

30. Chen G, Guan N, Huang K, Yi W. Fault-tolerant real-time tasks scheduling with dynamic fault handling. J Syst Architecture. 2020; 102: 101688. doi: 10.1016/j.sysarc.2019.101688.

31. Bharathi S, Chervenak A, Deelman E, Mehta G, Su MH, Vahi K. Characterization of scientific workflows. In: Third workshop on workflows in support of large-scale science. 2008: 1-10. doi: 10.1109/WORKS.2008.4723958.

Corresponding author
Fei Xie can be contacted at: fx439@uowmail.edu.au