Cluster Computing (2024) 27:9323–9344
https://doi.org/10.1007/s10586-024-04419-1

DFARM: a deadline-aware fault-tolerant scheduler for cloud computing

Ahmad Awan[1] · Muhammad Aleem[1] · Altaf Hussain[2] · Radu Prodan[3]

Received: 22 November 2023 / Revised: 11 February 2024 / Accepted: 5 March 2024 / Published online: 20 April 2024

Abstract

Cloud computing has become popular for small businesses due to its cost-effectiveness and the ability to acquire necessary on-demand services, including software, hardware, network, etc., anytime around the globe. Efficient job scheduling in the Cloud is essential to optimize operational costs in data centers. Therefore, scheduling should consider assigning tasks to Virtual Machines (VMs) in a Cloud environment in such a manner that could speed up execution, maximize resource utilization, and meet users' SLA and other constraints such as deadlines. For this purpose, the tasks can be prioritized based on their deadlines and task lengths, and the resources could be provisioned and released as needed. Moreover, to cope with unexpected execution situations or hardware failures, a fault-tolerance mechanism could be employed based on hybrid replication and the re-submission method. Most of the existing techniques tend to improve performance. However, their pitfall lies in certain aspects such as either those techniques prioritize tasks based on a singular value (e.g., usually deadline), only utilize a singular fault tolerance mechanism, or try to release resources that cause more overhead immediately. This research work proposes a new scheduler called the Deadline and fault-aware task Adjusting and Resource Managing (DFARM) scheduler, the scheduler dynamically acquires resources and schedules deadline-constrained tasks by considering both their length and deadlines while providing fault tolerance through the hybrid replication–resubmission method. Besides acquiring resources, it also releases resources based on their boot time to lessen costs due to reboots. The performance of the DFARM scheduler is compared to other scheduling algorithms, such as Random Selection, Round Robin, Minimum Completion Time, RALBA, and OG-RADL. With a comparable execution performance, the proposed DFARM scheduler reduces task-rejection rates by 2.34–9.53 times compared to the state-of-the-art schedulers using two benchmark datasets.

Keywords Cloud computing · Scheduling · Computation-aware scheduling · Fault tolerance · Cloud simulation

1 Introduction

Cloud computing allows organizations to easily utilize a scalable and elastic IT infrastructure through the Internet. Third-party companies often supply these services through *Service Level Agreements* (SLAs) that ensure the availability of distributed resources with required quality service for high-performance computing [1, 2]. To meet the SLAs and job-specific constraints such as execution deadlines, the *Cloud Service Provider* (CSP) must ensure that the data-center resources are ready and available around the clock, support an efficient job scheduling mechanism to prioritize tasks [3], and provide fault-tolerant execution.

Cloud job scheduling allocates tasks/cloudlets to the computing resources, a well-known NP-complete problem [1]. Many algorithms can effectively schedule tasks among multiple *Virtual Machines* (VMs) in a data-center [1, 4, 5], unscheduled downtime due to hardware/software or power failures compromising the efficiency of these schedulers. Some users may have specific execution deadlines, too, that require a scheduler capable of mapping the tasks to computing resources so that the specified deadlines are met. Designing a scheduler capable of efficiently mapping

✉ Muhammad Aleem
m.aleem@nu.edu.pk

[1] Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan

[2] Department of Computer Science, KICSIT Kahuta Campus, Institute of Space Technology, Islamabad, Pakistan

[3] Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria

the cloudlets and handling the machine failures by reallocating cloudlets to alternate machines is crucial in Cloud data centers. A scheduler can salvage work from a machine that must be taken offline for maintenance or has experienced a random fault by transferring it to a functioning machine in a data center [6]. Such fault-tolerant techniques help maintain SLAs seamlessly and minimize user impact during perceived downtime. Although task scheduling has received significant attention, the effects of machine failure for Cloud schedulers require increased attention. Machine failures can disrupt mission-critical or data-intensive applications, resulting in the loss of reputation, time, and money for the CSPs and their users. A fault-tolerant mechanism can proactively or re-actively move the affected machines to the functioning ones.

In addition to the fault tolerance aspect, users may have specific deadlines for completing their tasks and may prioritize them based on their urgency level. To meet these deadlines, the scheduler must efficiently assign tasks and aim to complete as many tasks as possible before the deadline. The tasks meeting their deadlines are marked as scheduled tasks, while the tasks failing their deadlines are marked as rejected tasks. However, fault tolerance and meeting deadlines can be conflicting objectives, as fault tolerance reduces the number of available resources while meeting deadlines requires sufficient resources. To balance the opposing goals, a scheduler must carefully consider the objectives and find a way to meet them effectively.

Existing techniques provide various solutions to the problem of optimizing deadline-constrained Cloud scheduling. Several techniques, such as EFDTS [3], DCCP [7, 8], and OG-RADL [9] incorporate dynamic resource allocation and deallocation that exploit cloud elasticity. Dynamic resources are vital components to improve performance while reducing cost. However, multiple solutions do not account for the boot and resource allocation times. Both allocation and deallocation are important because a resource in need if deallocated previously, could incur notable overheard if reallocated again. Some other solutions, such as RALBA [1, 8], and DEAS [10], either sort tasks by the task length or the deadline of the task. However, this simple and effective idea can make substandard decisions due to the simple heuristic based on the task length compared to the deadline for completing the task. The TCC [11] and EFDTS [3] employ a singular fault tolerance mechanism to counter runtime issues and other bottlenecks. Employing fault tolerance improves reliability, but in a deadline-constrained environment, the fault tolerance could reduce the usable number of resources (as some resources are reserved for realizing the fault-tolerance mechanism), resulting in more deadline misses.

To cope with these challenges, a scheduling algorithm called *Deadline and Fault-aware task Adjusting and Resource Management* (DFARM) is presented that efficiently distributes deadline-aware tasks and provides fault tolerance. The DFARM proposes a dynamic heuristic by establishing a relationship between task deadlines and size to prioritize them and determine the appropriate fault tolerance method. The DFARM schedules tasks on the VMs that can be completed faster. If a task cannot fulfill its deadline, DFARM reworks the queue to adjust the deadline-constrained task to an appropriate location without affecting other tasks' deadlines or incurring additional resources. Additionally, it dynamically acquires and releases resources for any task if necessary. The release mechanism allows the DFARM to free up the additional resources while minimizing the boot time if the resource acquisition is again necessary. The core objective of the DFARM scheduler is to reduce task rejection while providing fault tolerance with reduced resource provisioning. The main contributions of this work are:

- A twofold fault tolerance mechanism that selects the appropriate method based on the tightness of the deadline and task duration;
- A scheduling algorithm based on `adeadline per length of tasks` to prioritize tasks and employ appropriate fault tolerance mechanisms;
- A task adjustment technique to shuffle task queues and reduce the rejection rates;
- An efficient resource management method to minimize the boot time overhead while releasing the VMs;
- Task rejection rate performance results in an average of 2.34 and 9.53 times lower rejection rate (using two benchmark datasets, i.e., Synthetic and GoCJ) as compared to the state-of-the-art schedulers, respectively.

Section 2 discusses the state-of-the-art approach. Section 3 presents the DFARM scheduler, its design, algorithm, and time complexity analysis. Section 4 presents the experimental setup and performance evaluation, while Sect. 5 discusses the attained performance. Finally, Sect. 6 concludes the paper.

2 Literature review

Cloud scheduling is important for managing workloads in a data center, especially in a distributed and heterogeneous environment. Researchers have focused on two key areas: *fault tolerance* and *deadline-constraint scheduling*. Fault tolerance is the ability of the system to handle unexpected failures or errors without disrupting the overall operation, classified into proactive fault tolerance, which tries to anticipate and prevent problems before they occur, and reactive fault tolerance, which responds to faults and

attempts to recover from them. Deadline scheduling, however, involves tasks that must be completed within a specified time frame. If a task cannot be completed before the deadline, the system may reject or accept them depending on its nature. In hard real-time systems, missed deadlines are failures requiring task rejection, while in soft real-time systems, missed deadlines are acceptable if the task is eventually completed.

RALBA [1] is a cloud scheduling algorithm that prioritizes execution time by evenly distributing tasks among machines. It uses a Fill and Spill strategy to distribute tasks evenly across available VMs, according to their VMShare. In the Fill section, the scheduler assigns the largest possible tasks to VMs until no more VMShare is available. The Spill section assigns the remaining tasks to the VMs that can complete them at the earliest. RALBA is a dynamic scheduler that can reduce overall makespan and improve average resource utilization but that processes tasks in batches rather than just in time, unsuitable for time-sensitive applications and environments and other related workloads.

In [7], authors proposed a Cloud scheduler for deadline-constrained workflows on dynamically provisioned resources using Proportional Deadline-Constrained (PDC) and Deadline-Constrained Critical Path (DCCP) algorithms considering task dependencies, financial costs, and bandwidth costs. The authors pre-process the tasks into a bag of tasks spread across levels with no dependencies. The PDC or DCCP algorithms utilize the Cost Time Trade-off Factor (CTTF) and backfilling to optimize the pre-processed tasks further. The backfilling mechanism helps to utilize the gaps between dependent tasks and run on separate VMs, while CTTF considers the trade-off between cost and time. While this design considers the ability to acquire resources dynamically, it does not focus on releasing unnecessary resources to reduce costs.

The Heuristic-based load-balancing algorithm (HBLBA) [12] for Infrastructure as a Service (IaaS) clouds aim to configure servers efficiently based on the number and size of the incoming tasks using a rule set to determine the suitable VMs for task assignment. The rule set includes five rules that determine whether to create a VM instance on a machine with free resources or to add the tasks to a waiting queue until a VM becomes available. After the server configuration, the HBLBA maps the task to the VMs, creating fixed and dynamic queues for the host machine and VMs. The length of the queues depends on the CPU utilization and prioritizes the tasks based on a first-come, first-serve policy. While this approach can lead to better utilization of resources, it may result in an overall under-utilization by working on task batches.

In [8], the authors proposed a deadline-constrained dynamic load-balancing algorithm for scheduling tasks in a cloud environment. The algorithm focuses on handling load balancing with elasticity, allowing tasks that cannot meet their deadlines to be placed on dynamically created VMs to reduce the task rejection rate. The algorithm first sorts the tasks by their deadlines in ascending order. It then uses the matchmaker and dynamic task scheduler components to match tasks to the resources that can execute them quickly. The proposed elastic load balancing then checks the utilization of each VM and categorizes them as over-loaded, under-loaded, or balanced. Based on the average number of missed deadlines, the algorithm dynamically provisions and de-provisions a certain percentage of VM resources based on the average count of missed deadlines. While this approach may increase user satisfaction and reduce the missed deadline rate, it may result in missed deadlines, which can be prevented by provisioning resources at the time of need rather than waiting for a certain percentage of missed deadlines.

Efficient job scheduling is important for reducing power use in clouds. In [3], the authors proposed an energy-efficient algorithm called the Energy-aware Fault-Tolerant Dynamic Scheduling scheme (EFDTS) that optimizes resource utilization and energy consumption while scheduling jobs. The EFDTS algorithm divides tasks into categories and allocates them to the most suitable machines for scheduling efficiency while using replication to handle job failure. The algorithm also includes a migration policy to prevent over-utilization and turn idle machines off to save power. While this approach is effective, it may not be suitable for smaller data centers with few resources and flexibility to acquire additional resources dynamically. Therefore, arranging the new resources in these systems could result in more missed deadlines.

In [11], the authors focus on addressing Byzantine faults, a type of error that can go unnoticed in the early stages and of the workload execution cause significant damage, particularly when handling large data sets or mission-critical tasks in data centers. To address this issue, the authors use two algorithms: the Tactically Coordinated Checkpointing (TCC) algorithm, which is a proactive fault-tolerant system that eliminates faults before these incur an impact, and the Workload Sensitive Server Scheduling (WSSS) algorithm, which efficiently allocates resources. The TCC algorithm monitors the nodes and measures the delay in starting a virtual machine with a previous checkpoint to detect Byzantine faults, often indicated by delays. The WSSS algorithm tries to minimize the overhead caused by checkpointing. While these algorithms perform well in specific scenarios, checkpointing can be data-intensive. Storing delayed checkpoints on a remote backup server for fault tolerance may result in false positive flags for byzantine faults.

Deadline and Energy-aware Scheduling (DEAS) [10] algorithm schedules tasks within their deadlines. It begins by adding the set of newly arrived tasks and their deadlines to the Unbounded Queue. It then schedules them to the virtual machine with the minimum energy consumption if they complete before their deadlines and adds them to a Bounded Queue. If a task cannot be mapped to any virtual machine that can finish it before its deadline, the algorithm tries to schedule it immediately on any virtual machine that can finish it without violating the deadlines of the scheduled tasks. Otherwise, it provisions a new resource with the minimum MIPS required. After a certain threshold, it de-provisions it using dynamic voltage and frequency scaling. While this algorithm tries to schedule tasks efficiently, it does not consider the possibility of completing a task within the deadline by scheduling it within the queue rather than only at the head or tail.

As detailed in the provided text, the Overall Performance-based Resource-aware Dynamic Load-balancer (OG-RADL) [9] algorithm is designed for cloud environments with a specific focus on deadlines-constrained tasks. OG-RADL, in conjunction with the S-Scheduler, is responsible for assigning tasks to virtual machines based on their ability to complete them quickly within specified deadlines. The S-Scheduler intervenes when tasks cannot meet their deadlines on the initially assigned virtual machine, attempting to rearrange tasks to meet the timeline. The algorithm evaluates its performance using the Overall Gain normalization technique, which combines metrics such as ARUR, makespan, task rejection ratio, and task response time. However, OG-RADL is noted to have limitations, including the absence of a virtual machine provisioning and de-provisioning system for rejected tasks and potential challenges in rearranging tasks based on their deadlines, especially in linear deadline sort orders (Table 1).

3 Deadline and fault-aware task adjusting and resource managing (DFARM) scheduler

Figure 1 shows the system architecture of the DFARM scheduler comprising four layers: task queue, management layer, virtual layer, and physical layer.

3.1 System architecture

In a cloud computing environment, tasks arrive in a task queue through the system accessibility layers, such as a graphical, command line, or application programming interface. The scheduler module in the management layer then assigns each task in the queue to a VM in the management layer. If the task is completed before its deadline on a VM, it allocates that VM at the end of the scheduling queue. If the deadline is feasible, the scheduler checks if it can move the task up in the queue without affecting the other scheduled tasks. If such an execution pattern is impossible, the resource manager may provision an additional VM to execute the task. The scheduler marks the task rejected if no VM is available or provisioned in time. After assigning a task to a VM, it moves to the virtual layer, where all the VMs reside. The physical layer contains the machines on which the VMs run and execute the tasks. Each physical machine or host can host multiple VMs.

3.2 DFARM architecture

The core scheduling characteristic of a Cloud scheduler is to map tasks to the most suitable VM if possible or to find a sub-optimal placement of the tasks. Considering the generic guidelines, DFARM typically employs similar scheduling patterns abbreviations in the execution flow listed in Table 2.

The DFARM scheduling algorithm selects tasks in a queue with the minimum deadline per length as modeled in Eq. (16), prioritizing compute-intensive tasks with tight deadlines. The scheduler then calculates the completion time for the task on each VM and assigns it to the VM with the shortest completion time that can complete this task within the specified deadline. If no suitable VM is available, the scheduler uses an alternate strategy (i.e., repositioning the task in task queues) to assign the task. If the deadline still cannot be met using the alternate scheduling approach, the task is labeled as rejected.

$$\begin{aligned}\text{Deadline per length} &= \frac{\text{Deadline of the task}}{\text{Length of the task}} \\ &= \frac{Task.deadlineTime - Task.arrivalTime}{Task.MI}\end{aligned} \tag{1}$$

Before marking a task as rejected, the DFARM scheduler attempts two operations in sequence to possibly find the desired mapping. The *adjustment function* is related to the repositioning strategy, i.e., moving the task higher up in the task queue on any available VM, given that it does not affect the deadlines of the other scheduled tasks. This method assumes that the given task has a stricter deadline than the previous tasks and aims to execute it before some of the already scheduled tasks, considering the change does not result in a deadline violation. To determine the feasibility of this strategy, the scheduler calculates the new completion time for both the given task and the scheduled task. If both deadlines (with new completion timings) are satisfied, a reposition or swap is performed.

Table 1 Literature review summary

Algorithm	Methodology	Performance objectives	Limitations
RALBA [1]	– Fill and spill scheduler – Distribute tasks evenly w.r.t makespan	– Resource utilization – Makespan – Throughput	– A batch dynamic scheduler – Non-deadline aware – No fault tolerance
EFDTS [3]	– Deadline-aware dynamic scheduler – Classifies tasks and machines – Employs a reactive replication	– Migration with reduced power – Elastic resource provision	– Re-scheduling is not resource-aware – Assumes indefinite resources
TCC [11]	– Ranked mission-critical jobs – Measures delay between nodes – Kernel level checkpointing	– Detects byzantine faults	– No support for deadline-constraint tasks – Kernel-level checkpoint requires large storage
DCCP [7]	– Preprocesses tasks into a bag of tasks – Separate tasks into levels and sublevels – Optimizes by cost ratio and backfilling	– Transfer duration – Backfilling utilizes idle time – Sub-deadline-aware scheduler	– Lacks fault tolerance – Acquired VMs not released
OG-RADL [9]	– Assign tasks to VM with minimum completion time – If the deadline cannot be met, send to S-Scheduler – Presents a new normalization technique	– VM provisioning is used to reduce the rejection rate – Re-arrange tasks for better utilization	– Rather than finding a greater deadline in S-Scheduler, distance to deadline per task is employed
HBLBA [12]	– Provides dynamic scheduling configuring VMs based on rule set – Task-VM mapping is done on FCFS	– VM configuration to maximize resource utilization	– VM creation between batches may cause delay
[8]	– Sorts tasks by the deadline – Matchmaker and dynamic task scheduler component allocate tasks – Dynamically, resources management missed deadlines on average	– Load balancing is provided – Uses elasticity to reduce deadline misses	– Provisioning VM immediately would result in a lower miss rate
DEAS [10]	– Sort by the deadline and select VM by energy consumption – Resources management on energy consumption	– Based on energy consumption	– Task allocations for immediate execution may not work – Provisioning delay may result in missing the deadline
HunterPlus [13]	– Convolutional Neural Networks (CNN) based optimization of task scheduling	– Energy consumption per task – Task completion rate	– Non-fault tolerant technique – Non-deadline oriented tasks
Energy-aware resource provisioning [14]	– Multi-criteria scheduling based on fuzzy AHP-TOPSIS hybrid methodology	– Energy consumption – Response time – Resource utilization – User satisfaction	– Non-fault tolerant technique – Non-deadline oriented tasks
ANFIS [15]	– Fault-tolerant prediction model to proactively control resource load fluctuation	– Energy consumption – Task fault ratio – Resource utilization – Makespan and total cost	– Non-deadline oriented tasks

The *adjustment function* has two variants. The first variant, a simple or greedy version, strictly enforces both deadline checks for each iteration. The second variant, a more aggressive or non-greedy version, enforces the deadline check of the scheduled task for each iteration but may relax it for the given task. This simple change in enforcing strictness leads to drastic changes in behavior. The simple variant, which requires the condition to be satisfied initially, may end the search prematurely. Meanwhile, the aggressive variant increases its search space, as the

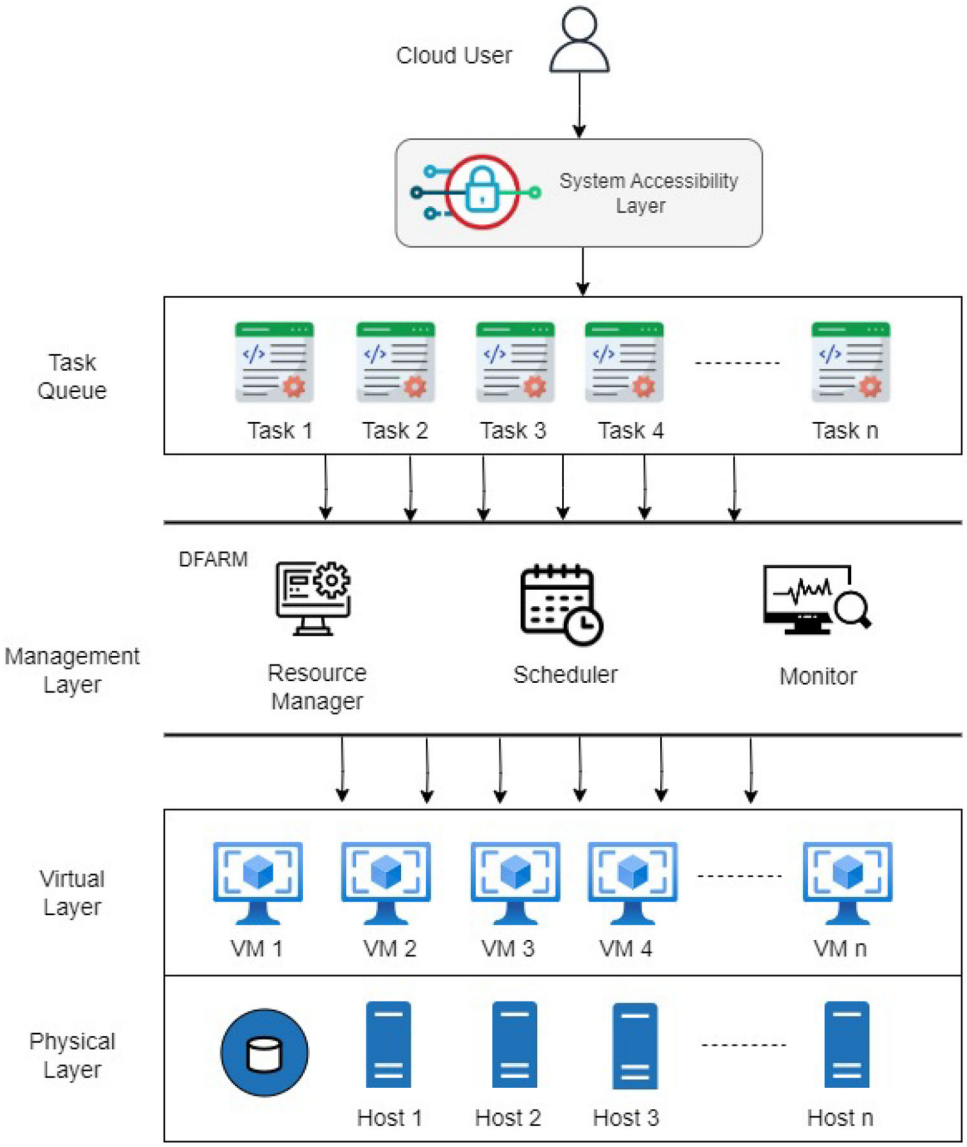


Fig. 1 Abstraction of DFARM system architecture

condition may be satisfied further down the list. However, while having a larger search space, the aggressive variant may still not find a suitable position, wasting time on a possible swap rather than guaranteeing a swap from the beginning.

The adjustment function determines the impact of introducing a new task in the scheduling queue by comparing the new completion time of the new and the already scheduled tasks on a particular VM. The function starts with the last task in the queue and calculates the revised completion time for both tasks (the already scheduled and the newly arrived task). If the revised completion time for both of the tasks (i.e., the newly arrived task and the already scheduled tasks) is within their corresponding deadlines then these tasks are swapped. If the scheduler does not find a suitable task to swap with, the DFARM scheduler provisions a new resource to meet the execution deadline. The DFARM scheduler ensures that the tasks with stricter deadlines are prioritized and placed higher up in the queue for execution.

There are both simple and aggressive versions of the adjustment function. In both versions of the adjustment function, the revised completion time of the scheduled task must not be affected, meaning it must still finish its execution before its deadline. However, the revised completion time of the new task only affects the function's behavior depending on the version. The simple version requires that, for each task in the scheduling queue, the function immediately returns the furthest scheduled task i on a VM where the new task's revised completion time can be finished before its deadline, starting from the end of the queue. This check must be valid from the beginning and at each step; otherwise, the function will either return

Table 2 Notation used and its description

Notations	Description
Task	The basic name for a job
Cloudlet	Notation for a job in CloudSim Simulator
MI	Million Instructors, size of Cloudlet in CloudSim
MIPS	Million Instructions Per Second, Computing Power of VM in CloudSim
$Clouet\_ET_{ij}$	Execution time of $Cloudlet_i$ on VM_j
$Cloudlet\_CT_{ij}$	Completion time of $Cloudlet_i$ on VM_j
VM_CT_j	Completion time on VMj
Cloudlet.MI	Size of task in MI on CloudSim
VM.MIPS	Computing power of VM in MIPS on CloudSim
TRR	Ratio of tasks rejected to all tasks
TAR	Ratio of tasks scheduled to all tasks
Task.arrivalTime	Time at which task is added to the scheduled queue
Task.deadlineTime	Deadline time after which a task would be considered rejected
Deadline per length	Ratio of deadline time to the length of task
$Cloudlet_CTscheduled_j$	Completion time of an already scheduled Cloudlet on VM_j
$Cloudlet_ETgiven_j$	Cloudlet execution time-adjusted to the scheduled queue on VM_j
$Cloudlet_ETscheduled_j$	Execution time for a Cloudlet already scheduled on VM_j
$newCloudlet_CTscheduled_j$	Updated completion time for an already scheduled Cloudlet on VM_j
$newCloudlet_CTgiven_j$	Cloudlet completion time-adjusted to scheduled queue on VM_j
DT	The deadline for the task
CT	Completion Time is calculated for executing a task on a VM
BT	The boot time after acquisition
AD	Acquisition delay before the resource is acquired

the task or move on to the next scheduled queue VM if the check fails at the start. The aggressive version performs the same check but allows the deadline check for the new task to fail to search further up the queue. If both conditions are met, the aggressive function will choose the scheduled task of VM j for the adjustment.

To illustrate the behavior of the adjustment function, we will consider an example where, given a list of scheduled tasks, what will be the outcome of adjusting a new task? For example, we have four tasks already scheduled on a VM V, namely tasks [$T3$, $T2$, $T1$, $T0$] where $T3$ is set for immediate execution, followed by $T2$, $T1$, and $T0$. Given the arrival of a new task Tx, the task Tx would have been scheduled to run after $T0$, but the deadline for Tx was not being satisfied by placing it at the end of the list, which prompted the adjustment function to position it higher within the list. Following are the outcomes of the adjustment function given certain scenarios.

- If the task Tx deadline could not be satisfied by scheduling it to run before $T0$, the adjustment function would fail and consider another VM.
- If the task Tx deadline is satisfied by scheduling it to run before $T0$, but $T0$ own deadline is violated, the adjustment function would fail and consider another VM.
- If the both task $T0$ and Tx deadline are satisfied by scheduling Tx to run before $T0$, the simple version of the adjust functions will adjust Tx to VM V schedule queue. Now that VM V has been selected for adjustment, it will repeat the deadline check process for all subsequent tasks scheduled to run on VM V until it cannot. Starting the search from $T0$ till $T3$, if task $T2$ deadline would be violated, the adjustment function would stop and perform the adjustment so Tx would be scheduled to run before $T1$ where the final sequence order would be [$T3$, $T2$, Tx, $T1$, $T0$].
- If the task Tx deadline could not be satisfied by scheduling it to run before $T0$, but $T0$ own deadline is satisfied even with adjustment, the aggressive version of the adjust functions can adjust Tx to VM V schedule queue. Now that VM V has been selected for possible adjustment, it will repeat the deadline check process for all subsequent tasks scheduled to run on VM V until it cannot. Starting the search from $T0$ till $T3$, the aggressive function would try to find a position where both deadline checks are satisfied, not just the first check. If Tx deadline cannot be satisfied anywhere, or if any scheduled tasks deadline is violated before both deadline checks can be satisfied, the adjustment function will fail. If both deadline checks were valid until

*T*2, the aggressive adjustment function could adjust task *Tx* where the final sequence order would be [*T*3, *Tx*, *T*2, *T*1, *T*0].

The execution time for a cloudlet i on a VM j is calculated as:

$$Cloudlet\_ET_{ij} = Cloudlet_i.MI / VM_j.MIPS \tag{2}$$

The completion time for a VM j is calculated as:

$$VM\_CT_j = \sum_{i=1}^{n} \left(\frac{Task_i.MI}{VM_j.MIPS} \right) \tag{3}$$

$$Cloudlet\_CT_{ij} = \frac{Cloudlet_i.MI}{VM_j.MIPS} + VM\_CT_j \tag{4}$$

The new completion time for the given and scheduled task is calculated for a VM j as:

$$newCloudlet\_CT_{\text{scheduledj}} = Cloudlet\_CT_{\text{scheduledj}} + Cloudlet\_ET_{\text{givenj}} \tag{5}$$

$$newCloudlet\_CT_{\text{givenj}} = Cloudlet\_CT_{\text{scheduledj}} - Cloudlet\_ET_{\text{scheduledj}} + Cloudlet\_ET_{\text{givenj}} \tag{6}$$

Only if the previous method fails does the scheduler move on to the next strategy, i.e., trying to provision a new resource (exploiting the Cloud elasticity). The new resource must be able to execute the task before the deadline while considering the time to acquire the resource (if applicable) and boot time.

If both methods fail, the task is added to the rejected task list, the user has the option to configure the scheduler to allow the task to run even after rejection. If the task is allowed to run after rejection, the scheduler becomes a "soft-line" scheduler, which may impact the execution of other tasks that have not yet been scheduled. On the other hand, if the task is not allowed to run after rejection, the scheduler becomes a "hard-line" scheduler, which is more suitable for mission-critical tasks that must meet the specified deadlines.

When the task queue is empty, and no more tasks are scheduled, the scheduler may attempt to reduce additional costs by releasing or de-provisioning the idle resources. This is beneficial because keeping resources available and idle results in continuous energy use and increased costs. However, there is a potential drawback to immediately releasing idle resources. For example, if a new task arrives shortly after the release of a resource, the scheduler may need to re-acquire the resource, incurring additional costs such as boot time and acquisition delay. To mitigate this issue, the DFARM scheduler lets the resources remain idle for a certain time before de-provisioning them. This allows the resources to be ready for additional work without incurring the costs related to resource re-acquisition while also avoiding the drawbacks of keeping idle resources provisioned for an extended time. Using this strategy, the proposed scheduler maintains a balance between the costs of releasing idle resources and re-acquiring them.

To attain fault tolerance, i.e., ensuring the task completion despite errors or failures. The proposed DFARM scheduler employs replication and resubmission methods depending on the task's deadline. To determine the appropriate fault tolerance strategy for a given task, its deadline per length (see Eq. (16), e.g., How many times the task can run again and again before the deadline expires) can be compared to the provenance data of the other tasks. This is attained by adding the given task's deadline per length to the provenance data (i.e., historical values) in a sorted manner. If the given task falls within a certain range, up to a point known as the *replication–resubmission ratio* (typically set between 0 and 1), the task's deadline is considered too strict, and the task is replicated. Replication involves immediately scheduling the task to run on a separate machine or VM from the original task. If the given task falls outside this range, the re-submission strategy is applied, which involves only rescheduling the task if the VM or running machine crashes and requires the task to be restarted. Using these fault tolerance measures, the cloud scheduler can ensure that tasks are completed even during failures or errors while also considering the strictness of the task's deadline and the relative cost of replication versus re-submission.

Using these methods, the proposed DFARM scheduler can efficiently assign tasks while trying to account for additional problems the other solutions face.

3.3 Dataset

The evaluation of the proposed DFARM scheduler is conducted using three datasets: *Google Cloud Jobs* (GoCJ) dataset [16], *Heterogeneous Computing Scheduling Problems dataset* (HCSP) [17], and *synthetic workload* [1]. The Google-like workload or the GoCJ dataset [16] contains tasks and the number of instructions generated using the Monte-Carlo simulation method based on realistic Google cluster traces. Using Google cluster trace analysis and MapReduce logs from the M45 supercomputing cluster, a workload of various sizes was generated ranging from 15,000 to 900,000 Million Instructions (MI).

The datasets do not contain any deadline information, which must be added through a separate process. Using the technique from [18], the deadline is calculated by adding the parameter *baseDeadline* to the arrival time of the task.

$$d_i = a_i + baseDeadline \tag{7}$$

where d_i is the deadline while a_i is the arrival time of the task.

The *baseDeadline* is generated using a uniform distribution $\bigcup(baseTime, a * baseTime)$ where *baseTime* varies from 100 to 400 with a step of 50 while a is set to 4.

3.4 Pseudo code of DFARM scheduler

The source code of the DFARM Algorithm is provided on https://github.com/amzshow/DFARM.

Algorithm 1 DFARM algorithm

Input vmList - list of VMs, taskList - list of tasks
Output taskVmMap - Mapping of tasks to VM

```
availableTask ⇐ tasksList
availableVm ⇐ vmList
rejectedTasks ⇐ LinkedList < task >
vmCtMap ⇐ Map < VM, double >
VmCtMap ⇐ getAllVmCt(tasksList, availableVm)
deadlinePerLengthHistory ⇐ LinkedList < Double >
replicateResubmitRatio ⇐ int(0 − 1)
while availableTask.size() ≠ 0 do
    taskIndex ⇐ findMinDeadlinePerLength(availableTask)
    task ⇐ availableTask.remove(taskIndex)
    replicateTask ⇐ determineFaultToleranceTechnique(task, deadlinePerLengthHistory, replicateResubmitRatio)
    if replicateTask and task.isDuplicate ≠ true then
        copyTask ⇐ task
        copyTask.isDuplicate ⇐ true
        availableTask ⇐ availableTask.prepend(copyTask)
    end if
    deadlinePerLengthHistory ⇐ deadlinePerLengthHistory.append(task.length/task.length)
    newVmCtSet ⇐ newVmCtSet.sort(value)
    selectedVm ⇐ assignAcquire(task, availableVm, newVmCtMap, newVmCtSet)
    if selectedVM = null then
        rejectedTasks ⇐ rejectedTasks.append(selectedVm)
        if ALLOW_TASK_TO_RUN then
            if len(vmCtSet) ≠ 0 then
                selectedVM, vmCt ⇐ vmCtSet.get(0)
            end if
        end if
    end if
    if selectedVM ≠ null then
        taskVmMap ⇐ taskVmMap.put(task, selectedVm)
        vmCt ⇐ vmCtMap.get(selectedVM) + (task.length/selectedVm.mips)
        vmCtMap ⇐ vmCtMap.put(selectedVm, vmCt)
    end if
    if availableTask.size() = 0 then
        if availableVm.size() = 0 then
            break
        else
            availableVm ⇐ releaseVm(availableVm, VmCtMap)
        end if
    end if
end while
returns taskVmMap
```

Algorithm 2 Find task with minimum deadline per length

Input tasksList - List of tasks
Output taskIndex - Index of the task with minimum deadline per length

```
taskIndex ⇐ 0
min ⇐ tasksList.get(0)
for index, task of tasksList do
    if task.deadline/task.length < min.deadline/min.length then
        min ⇐ task
        taskIndex ⇐ index
    end if
end for
returns taskIndex
```

Algorithm 3 Get All VM completion time

Input tasksList - List of tasks, vmList - List of VM to assign task
Output vmCtMap - Completion time for task on all VM

```
for vm of vmList do
    ct ⇐ vm.bootDelay + vm.acquisitionDelay
    for task of tasksList do
        if task.assignedVm = vm.id then
            ct ⇐ ct + (task.length/vm.mips)
        end if
    end for
    vmCtMap ⇐ vmCtMap.put(vm, ct)
end for
returns vmCtMap
```

Algorithm 4 Assign and acquire

Input task - selected task, availableVm - list of VMs available, vmCtMap - Completion time for the task on all VM, vmCtSet - Set of Completion time for all VM sorted by completion time
Output availableVm - list of VMs available

```
selectedVm ⇐ null
for vm, vmCt of vmCtSet do
    if vmCt + (task.length/vm.mips) ≤ task.deadline then
        selectedVm = vm
        break
    end if
    candidate ⇐ adjustPossible(task, vm, vmCtMap)
    if candidate ≠ null then
        adjustToVm(task, vm, candidate)
        selectedVm = vm
        break
    end if
end for
if selectedVm = null then
    selectedVm ⇐ acquireVm(task, vmCtMap)
    if selectedVm ≠ null then
        availableVm ⇐ availableVm.append(selectedVm)
    end if
end if
returns availableVm
```

Algorithm 5 Acquire VM

Input task - task to search VM for, vmCtMap - Completion time for task on all VM
Output newVM - newly acquired VM

$newVm \Leftarrow null$
$tempVmCtMap \Leftarrow Map < VM, double >$
$minVmCt \Leftarrow null$
for `vm for allAvailableVm` **do**
  $vmCt \Leftarrow vm.bootTime + vm.aquisitionDelay(task.length/vm.mips)$
  if $vmCt \leq task.deadline$ **and** $(minVmCt = null$ **or** $vmCt \leq minVmCt)$ **then**
    $newVm \Leftarrow vm$
    $minVmCt \Leftarrow vmCt$
    break
  end if
end for
$vmCtMap \Leftarrow vmCtMap.put(newVm.id, newVm.bootTime + newVm.acquisitionDelay + (task.length/newVm.mips)$
returns $newVm.acquire()$

Algorithm 6 Release VM

Input availableVm - list of VMs available, vmCtMap - Completion time for the task on all VM
Output availableVm - list of VMs available

for `vm for availableVm` **do**
  $vmCt \Leftarrow vm.bootTime + vm.acquisition$
  if $vmCt + vm.bootTime + vm.aquisitionDelay < NOW()$ **then**
    $availableVm \Leftarrow availableVm.remove(vm)$
    $vm.release()$
  end if
end for
returns $availableVm$

Algorithm 7 Check if the adjustment is possible

Input task - Task to assign, vm - VM to assign task, vmCtMap - Completion time for task on all VM, deepSearch - perform deep search, default False
Output selectedTask - Task available for adjustment

```
taskList ⇐ getTaskList(vm)
vmCt ⇐ vmCtMap.get(vm)
taskExecTime ⇐ task.length/vm.mips
selectedTask ⇐ null
for candidate of taskList do
    candidateExecTime ⇐ candidate.length/vm.mips
    newCandidateVmCt ⇐ vmCt + taskExecTime
    newTaskVmCt ⇐ vmCt − candidateExecTime + taskExecTime
    newTaskSatisfied ⇐ newTaskVmCt ≤ task.deadline
    newCandidateSatisfied ⇐ newCandidateVmCt ≤ candidate.deadline
    if deepSearch then
        if newCandidateSatisfied then
            vmCt ⇐ vmCt − candidateExecTime
            if newTaskSatisfied then
                selectedTask ⇐ candidate
            end if
        end if
    else
        if newTaskSatisfied and newCandidateSatisfied then
            selectedTask ⇐ candidate
            vmCt ⇐ vmCt − candidateExecTime
        else
            break
        end if
    end if
end for
returns selectedTask
```

Algorithm 8 Determine the fault-tolerance technique

Input task - Task to assign, deadlinePerLengthHistory - List of deadline/length of previous tasks, replicateResubmitRatio - Float value to determine how much percentage of the minimum values of deadline/length history it needs to be in
Output replicationRequired - True for replication otherwise False for resubmission

```
if history.size() ≤ 0 then
    returns true
end if
replicationRequired ⇐ true
index ⇐ replicateResubmitRatio ∗ (deadlinePerLengthHistory.size() − 1)
value ⇐ task.deadline/task.length
if value ≥ deadlinePerLengthHistory.get(index) then
    replicationRequired ⇐ true
end if
returns replicationRequired
```

3.5 Complexity and overhead analysis

This section describes the time complexity analysis of the DFARM scheduler as compared to the other approaches. For analysis, we will consider N as the total number of tasks that need to be scheduled and M as the total number of VMs where the N tasks will be mapped to M VMs. Concerning real-world cloud computing environments, we will consider $N > > M$. The proposed algorithm can be divided into two outer segments: (i) scheduling the task to a VM, and (ii) releasing the VM. The final release segment is a simple for loop that checks each VM to see if they have completed all their tasks and enough time has passed which has a time complexity of $O(N)$. Although, the scheduling segment is a simple for loop over the N tasks, the time complexity of the inner part of the for loop needs to be calculated, referred to as the inner segment. The inner segment can be divided into the following actions: (i) select the tasks by minimum deadline per length, (ii) find the VM with minimum completion time, (iii) Check if adjustment to the already scheduled VMs is possible, and (iv) acquiring new VM if the task cannot be assigned to any already acquired VM. For the first action of the inner segment, the scheduler iterates over the task queue to find a task with the minimum deadline per length that has a time complexity of $O(N)$. For the next action of the inner segment, the scheduler mimics the *Minimum Completion Time* or *MCT* algorithm by finding the VM with minimum completion time that can finish the task within the deadline, and the *MCT* algorithm has a time complexity of $O(NM)$, the time complexity for the proposed algorithm is $O(M)$ as we only need to iterate over the VMs. For the adjustment inner segment, the proposed algorithm iterates over the VMs M and the already scheduled tasks N_s (where N_s is a subset of N) to check if an adjustment is possible. If adjusting is possible, it will insert it in the correct position in the scheduled queue for the specific VM. In the final inner segment, the algorithm will try to acquire a new VM that can finish the task within the deadline which has a time complexity of $O(M_{available})$. Combining all the time complexity of the inner segments, we get a time complexity of $O(N + M + MN_s + M_{available})$. If we simplify it, so $N_s = N$ and $M_{available} = M$ then the time complexity would be $O(N + M + MN + M)$. If we combine the inner and outer segment equations, the time complexity would be $N(N + M + MN + M) + M)$ or $O(N^2 + MN + N^2M + NM + M)$, which could be simplified to $O(N^2 M)$ or $O(M.N^2)$.

For comparison, the proposed schedulers' time complexity is compared with Random Selection (RS), Round Robin (RR), Minimum Completion Time (MCT), RALBA [1] and OG-RADL [9] (please see Table 3).

4 Experimental evaluation and discussions

This section encompasses the experimental evaluation of the proposed scheduler DFARM and compares it with other scheduling heuristics.

4.1 Experimental setup

The solution is implemented in Java language and evaluated using the simulator CloudSim [19]. Cloudsim is a well-known open-source cloud simulation tool that lets us simulate real cloud environments and services for modeling and evaluation. The experiment is performed on an Intel(R) Core(TM) i5-10300 H CPU @ 2.50GHz with 16 GB of Main Memory and an Nvidia Geforce GTX 1650 with 4 GB of dedicated VRAM. The programming language used was Java running OpenJDK Runtime Environment Corretto−8.282.08.1 (build 1.8.0_282-b08). The project was run on Java from Eclipse version 2020-12 (4.18.0) running on Windows 10 Home version 20H2. Table 4 showcases the configuration detail used for the simulation. The experiments are performed with 30 host machines that manage 50 VMs in a data center. The VM's computation power is measured in Millions of Instructions Per Second (MIPS) as listed in Table 5.

4.2 Workload generation

For this experiment, two workloads are employed (i) a synthetic workload and (ii) a Google-like benchmark workload GoCJ. The cloudlet configuration used in synthetic workload is randomly generated by using different ranges for the cloudlets. The generated cloudlets are measured in Million Instructions (MI) and the ranges for the synthetic workload are listed in Table 6.

The GoCJ dataset [16] contains tasks and their number of instructions generated using the Monte-Carlo simulation method based on realistic Google cluster traces. Using

Table 3 Time complexity of various algorithms

Algorithm	RS [17]	RR [17]	MCT [1]	RALBA [1]	OG-RADL [9]	DFARM (proposed)
Complexity	$O(N)$	$O(N)$	$O(M.N)$	$O(M^2.N)$	$O(N^2)$	$O(M.N^2)$

Table 4 Configuration used for simulation

Item	Specification
Simulator/version	CloudSim version 4.0
Computing power of host machines	4 Dual-core (4000 MIPS)
	26 Quad-core (4000 MIPS)
Total host machines	30 Hosts
Host machine memory	16,384 MBs each
Total VMs	50 VMs
Total cloudlets	80 Cloudlets (synthetic workload)
	100 Cloudlets (Google-like workload)

Table 5 Computation power of VMs, measured in Million Instructions per Second

Number of VMs	Computational power (MIPS)
7	100
7	500
6	750
6	1000
6	1250
6	1500
6	1750
6	4000

Table 6 Size of cloud-lets for synthetic workload, measured in Million Instructions

Name	Number of cloudlets	Cloudlet size (MI)
Tiny	20	1–250
Small	60	800–1200
Medium	5	1800–2500
Large	10	7000–10,000
Extra large	5	30,000–45,000

Table 7 Size of cloudlets for Google-like workload, measured in Million Instructions

Name	Number of cloudlets	Cloudlet size (MI)
Small	20	15k–55k
Medium	40	59k–99k
Large	30	101k–135k
Extra large	4	150k–337.5k
Huge	6	525k–900k

Google cluster traces analysis and MapReduce logs from the M45 supercomputing cluster, the workload of various sizes is generated ranging from 15,000 to 900,000 MI described in Table 7.

As none of the datasets contain any deadline information, the deadline information is added using the technique from the study [18], and the deadline is calculated by adding the parameter *baseDeadline* to the arrival time of the task. The deadline is generated using a uniform distribution $\bigcup(baseTime, a * baseTime)$, where *baseTime* varies from 100 to 400 with a step of 50 while a is set to 4 for the GoCJ workload while the synthetic workload is set to 2 to 8, the same values as GoCJ however divided by 50.

4.3 Performance metrics

A cloud scheduler is evaluated on how efficiently and fairly it can schedule tasks amongst VMs, keeping that in mind, the DFARM scheduler has been evaluated using the following performance metrics: *Makespan* [1], *Throughput* [1], and *Average Resource Utilization Ratio* (ARUR) [1]. Besides the above metrics, the scheduler will also be evaluated using *Task Rejection Rate* (TRR) [3] and *Task Acceptance Rate* (TAR) to measure how many tasks the algorithm was able to schedule to meet the task deadlines.

Makespan metric measures the time taken by the slowest machine.

$$\text{Makespan} = \max_{\forall j \in \{1,2,3,\ldots,m\}} (VM\_CT_j) \tag{8}$$

where m is the number of VMs and VM_CT_j is the VM completion time of a VM calculated using

$$VM\_CT_j = \sum_{i=1}^{n} \left(\frac{Task_i.MI}{VM_j.MIPS} \right) \tag{9}$$

where n is the number of tasks. For VM_CT_j, n will be the number of tasks that are assigned to the VM j. Throughput measures the execution of the workload tasks per unit time.

$$\text{Throughput} = \frac{n}{\text{Makespan}} \tag{10}$$

ARUR measures the overall utilization of the resource and indicates how well a resource was utilized.

$$ARUR = \frac{\left(\frac{\sum_{j=1}^{m} VM\_CT_i}{m}\right)}{\text{Makespan}} \tag{11}$$

TRR and TAR measure the number of tasks that the algorithm was able to schedule within the deadline. The value for both metrics will always be between 0 and 1 and is an important metric for how well a deadline scheduler is performing.

$$TRR = \frac{\text{Number of Tasks Rejected}}{\text{Total Number of Tasks}} \tag{12}$$

$$TAR = \frac{\text{Number of Tasks Accepted}}{\text{Total Number of Tasks}} \tag{13}$$

The values indicate how well a deadline scheduler is performing. The TAR value should be as high as possible while the TRR should be as close to 0 as possible. Both values are a subset of the number of tasks such that their addition will return 1.

$$TAR + TRR = 1 \tag{14}$$

4.4 Simulation results

The proposed scheduling algorithm is compared with other cloud scheduling algorithms (i.e., Random Selection, Round Robin, Minimum Completion Time, RALBA [1], OG-RADL [9]) based on makespan, throughput, ARUR, and task rejection rate, on both synthetic and the GoCJ workload. The experiment is performed 100 times and the average values are reported here.

Along with the comparison of algorithms, data is also presented that showcases the effects of fault tolerance and VM reservation before the scheduler starts. Fault tolerance is determined by a 0–1 parameter or the replication–resubmission ratio (i.e., percentage of tasks to replicate, where 1 shows 100% replication). The replication–resubmission ratio has a notable impact on performance too, while the VM reservation shows the minimum number of VMs required at the start.

In addition to the comparison of the results, the different performance implications or versions of the proposed DFARM scheduler have been included to provide additional insight. The different versions of the DFARM scheduler are created by changing the default selection and whether a deep search is enabled for the adjustment function or not. The possible selection methods are: (i) No Sort (select the first item in the queue), (ii) Deadline Sort (select a task with a minimum deadline), (iii) Deadline/Length Sort (select a task with a minimum deadline per length), and (iv) Length/Deadline Sort (select a task with minimum length per deadline which is a reverse of deadline per length). Whether the deep search is enabled (indicated by `Deep (enabled)` label) for the *adjustment function* or not results in a total of eight (08) variants or versions of the proposed DFARM scheduler. The versions created using *No Sort* and *Deadline Sort* are used to showcase the importance of sorting/selection mechanisms and their impact, while the Length/Deadline shows the effects of bad sorting/selection.

The data for both synthetic and the GoCJ benchmarks shows a pattern emerging among the various versions of the scheduler. Repeatedly, the No Sort version exhibits the worst performance, with only certain outliers. On the other hand, the Deadline Sort version does improve performance, highlighting the significance of sorting in the scheduling process. However, it is noteworthy that while sorting generally improves performance, there are cases where it can decrease performance. The Length/Deadline sort version serves as an example of non-optimal or bad sorting, as it produces comparable performance to the No-Sort version and, in some instances, even underperforms. In contrast, the Deadline/Length sort version consistently outperforms all the other DFARM versions, establishing itself as the superior choice. Additionally, a consistent trend can be observed wherein the deep version always boosts performance. Although the performance increase can vary, the positive impact is persistent. Consequently, the Deadline/Length Sort version with the deep search enabled emerges as the clear choice for performance, making it the preferred variant of the DFARM versions for further comparisons and evaluation as compared to the other scheduling heuristics.

To provide a better context when comparing each performance metric, Eq. 17 is used to showcase the percentage of improvement of a scheduler x compared to the scheduler y.

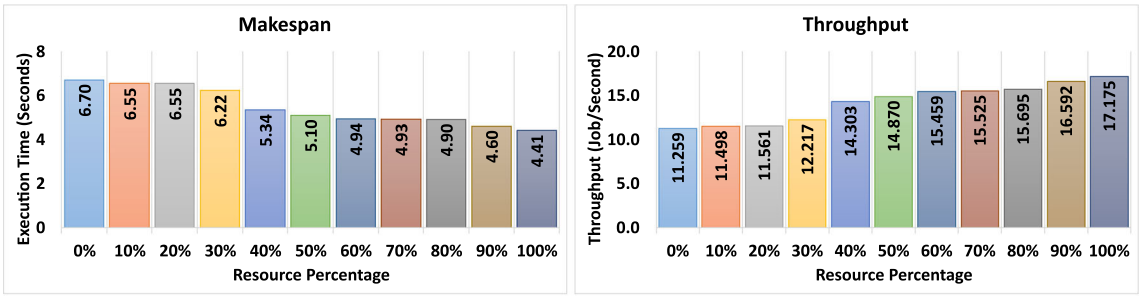
$$\begin{aligned}\text{Deadline per length} &= \frac{\text{Deadline of the task}}{\text{Length of the task}} \\ &= \frac{Task.deadlineTime - Task.arrivalTime}{Task.MI}\end{aligned} \tag{15}$$

$$\text{Length per deadline} = \frac{1}{\text{Deadline per length}} \tag{16}$$

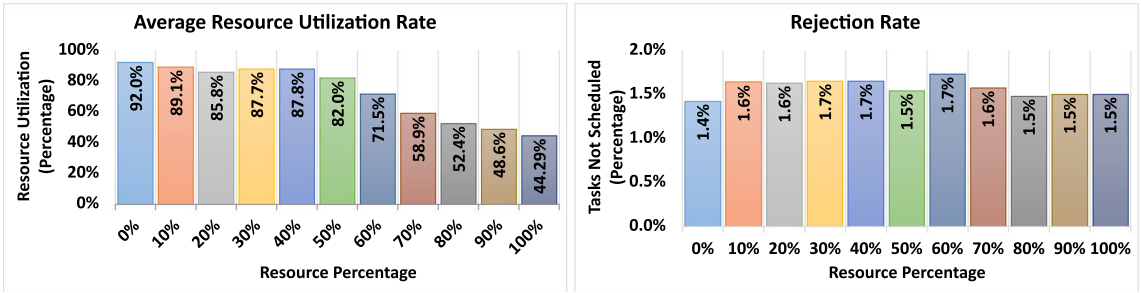
$$\text{Improvement} = \frac{x - y}{abs(y)} * 100 \tag{17}$$

4.4.1 Execution performance

Figure 2 shows the performance results for the synthetic workload while Fig. 3 shows performance results for the GoCJ benchmark workload. As shown in Fig. 2, the proposed DFARM scheduler outperforms the other heuristics such as RS, RR, MCT, and OG-RADL both in task rejection rate, only the RALBA scheduler slightly performs

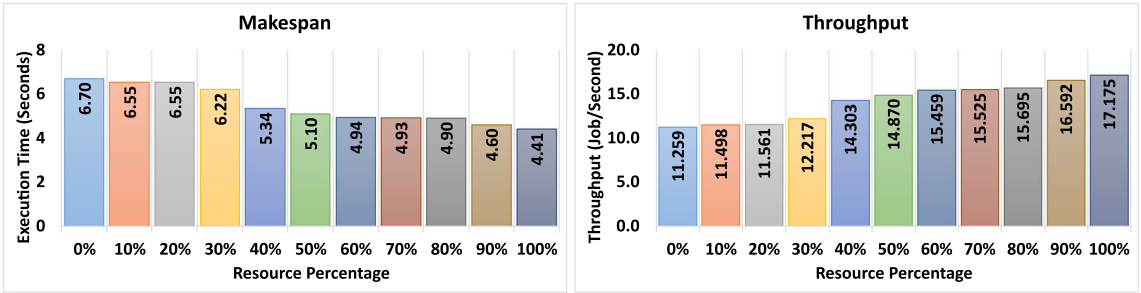


(a) Average Makespan for Synthetic Workload. **(b)** Average Throughput for Synthetic Workload.

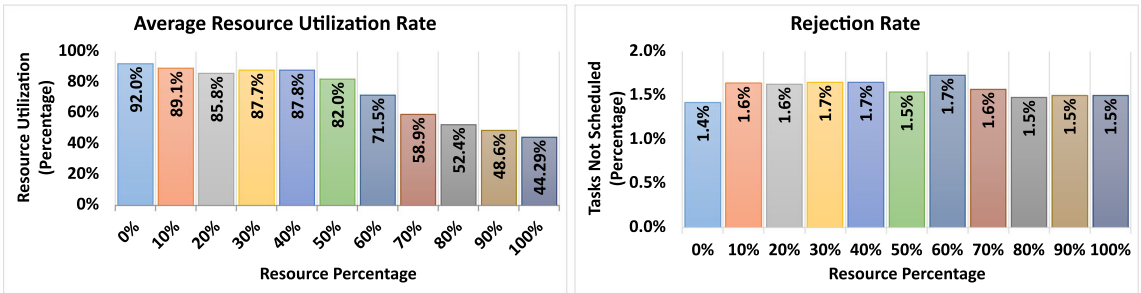


(c) Average ARUR for Synthetic Workload. **(d)** Average Rejection Rate for Synthetic Workload.

Fig. 2 Results for synthetic workload



(a) Average Makespan for GoCJ Workload. **(b)** Average Throughput for GoCJ Workload.



(c) Average ARUR for GoCJ Workload. **(d)** Average Rejection Rate for GoCJ Workload.

Fig. 3 Results for GoCJ workload

better. As Fig. 3 shows the DFARM scheduler has outperformed all the other scheduling algorithms in different performance aspects and produced a standout performance, only slightly being outperformed by the RALBA in makespan.

The makespan (i.e., the execution time for the full workload) for the synthetic dataset is depicted in Fig. 2a. The results shown in Fig. 2a depict that the RALBA scheduler attained 34.85%, 8.56%, 15.56%, 13.97%, 1.86% lower makespan as compared to RS, RR, MCT, OG-

RADL, and the DFARM scheduler, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 3c. These results show that the RALBA scheduler attained 47.23%, 17.27%, 46.47%, 44.42%, 0.07% lower makespan as compared to RS, RR, MCT, OG-RADL, and DFARM, respectively. These results show that the proposed DFARM scheduler has very similar makespan performance (a maximum of 1.86% more execution time as compared to the best-performing scheduler i.e., RALBA. However, the proposed DFARM scheduler outperforms significantly in the task-rejection aspect as compared to the other state-of-the-art schedulers (as shown in the experiments in Figs. 2d and 3).

The throughput (i.e., number of jobs executed per second) for the synthetic dataset is depicted in Fig. 2b. The results shown in Fig. 2b depict that the RALBA scheduler attained 37.12%, 21.96%, 17.05%, 16.17%, 2.29% higher throughput as compared to RS, RR, MCT, OG-RADL, and DFARM scheduler, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 3b. These results show that the DFARM scheduler attained 47.60%, 27.03%, 35.19%, 12.06%, 34.75% higher throughput as compared to RS, RR, MCT, RALBA, and, OG-RADL, respectively.

The ARUR (i.e., average resource utilization ratio) for the synthetic dataset is depicted in Fig. 2c. The results shown in Fig. 2c depict that the RALBA scheduler attained 41.51%, 16.02%, 52.36%, 52.35%, 1.64% higher resource utilization as compared to RR, RS, MCT, OG-RADL, and DFARM, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 3c. These results show that the DFARM scheduler attained 53.24%, 26.36%, 63.33%, 6.20%, 63.15% higher resource utilization as compared to RS, RR, MCT, RALBA, and OG-RADL, respectively.

The task rejection rate (i.e., number of tasks rejected) for the synthetic dataset is depicted in Fig. 2d. The results shown in Fig. 2d depict that the DFARM scheduler attained 3.45, 3.77, 1.6, 1.24, 1.67 × (i.e., times) lower rejection rates as compared to RS, RR, MCT, RALBA, and OG-RADL, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 3d. These results show that the DFARM scheduler attained 16.26, 10.73, 5.8, 9.06, and 5.8 × lower task-rejection rates as compared to RS, RR, MCT, RALBA, and OG-RADL, respectively.

4.4.2 Dynamic resource acquisition

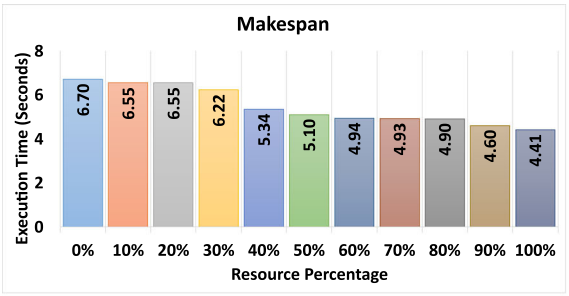
Dynamic resource acquisition is the minimum number of VMs required at the start of the workload execution or pre-allocations. This represents a percentage value from 0 to 100, where 0% indicates that no VMs are reserved at the start while 100% indicates that all the VMs are pre-allocated (at the start of the workload execution). Figure 4 shows the performance results for the synthetic workload execution while the execution of the GoCJ benchmark dataset has been depicted in Fig. 5. These experiments were conducted using several pre-allocation schemes such as 0%, 20%, 40%, 60%, 80%, and 100% prior availability of the required number of VMs. The purpose of these experiments was to gauge the impact of pre-allocation of the resources (i.e., VMs) on performance aspects such as makespan, throughput, ARUR, and task rejection rate.

The impact of dynamic resource allocation on the attained makespan by the DFARM scheduler using a synthetic dataset is presented in Fig. 4a. The results shown in Fig. 4a depict that the pre-allocation of VMs i.e., 0%, 20%, 40%, 60%, and 80% as compared to to 100% pre-allocation yields 51.85%, 48.44%, 20.98%, 11.99%, 11.07% lower makespan, respectively. In Fig. 5a, the makespan results of the execution of the GoCJ dataset depict that the pre-allocation of VMs i.e., 0%, 20%, 40%, 60%, and 80% as compared to 100% results in 66.18%, 67.08%, 61.58%, 43.28%, 17.49% lower makespan, respectively.

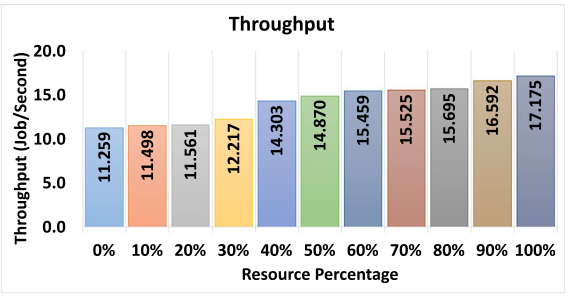
The impact of dynamic resource allocation on the attained throughput by the DFARM scheduler using a synthetic dataset is depicted in Fig. 4b. The results shown in Fig. 4b shows that the pre-allocation of VMs i.e., 0%, 20%, 40%, 60%, and 80% as compared to to 100% pre-allocation results in 34.44%, 32.69%, 16.72%, 9.99%, 8.62% higher throughput, respectively. In Fig. 5b, the throughput results of the execution of the GoCJ dataset show that the pre-allocation of VMs 0%, 20%, 40%, 60%, and 80% as compared to 100% results in 40.13%, 40.60%, 38.43%, 30.03%, 14.13% higher throughput, respectively.

The impact of dynamic resource allocation on ARUR attained by the DFARM scheduler using a synthetic dataset is depicted in Fig. 4c. The results shown in Fig. 4c shows that the pre-allocation of VMs i.e., 0%, 20%, 40%, 60%, and 80% as compared to to 100% pre-allocation results in 6.69%, 4.49%, 22.21%, 43.07%, 51.85% higher resource utilization, respectively. In Fig. 5c, the ARUR result of the execution of the GoCJ dataset show that the pre-allocation of VMs 0%, 20%, 40%, 60%, and 80% as compared to 100% results in 3.07%, 5.85%, 1.30%, 3.58%, 11.59% higher resource utilization, respectively.

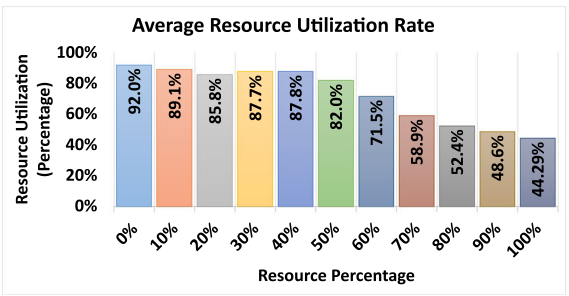
The rejection rate (i.e., number of tasks rejected) for the synthetic dataset is depicted in Fig. 4d. The results shown in Fig. 4d depict that pre-allocation at 80% attained 2.41%, 1.61%, 1.61%, 4.51%, 1.62% lower rejection rate as compared to 0%, 20%, 40%, 60%, and 100%, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 5d. These results show that pre-allocation at 0% attained 0.21%, 0.23%, 0.31%, 0.06%, 0.08% lower rejection rate as compared to 20%, 40%, 60%, 80%, and 100%, respectively.



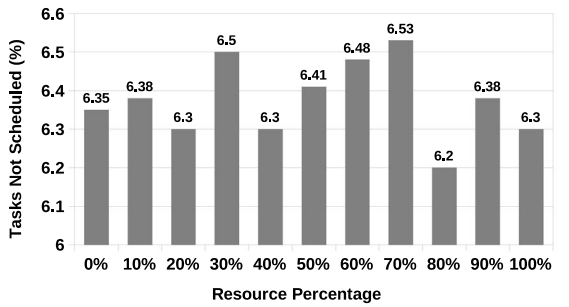
(a) Average Makespan for the percentage of VM reserved before the scheduler starts.



(b) Average Throughput for the percentage of VM reserved before the scheduler starts.

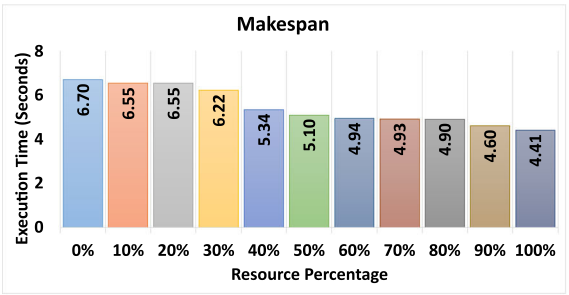


(c) Average ARUR for the percentage of VM reserved before the scheduler starts.

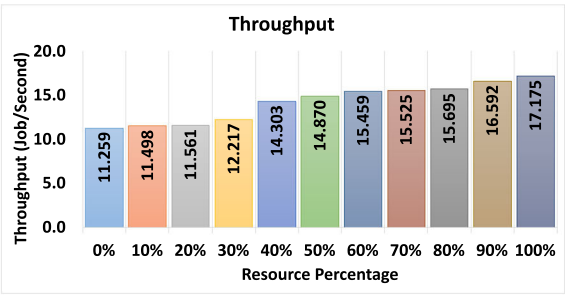


(d) Average Rejection Rate for the percentage of VM reserved before the scheduler starts.

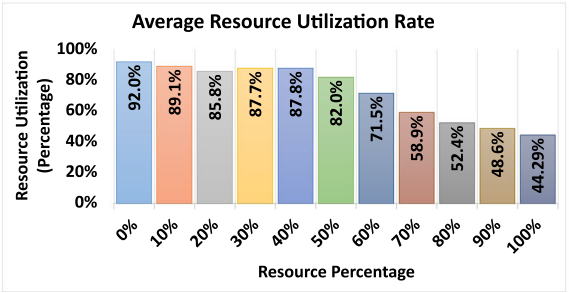
Fig. 4 Results for the percentage of VM reserved before the scheduler starts on Synthetic Workload using DFARM scheduler



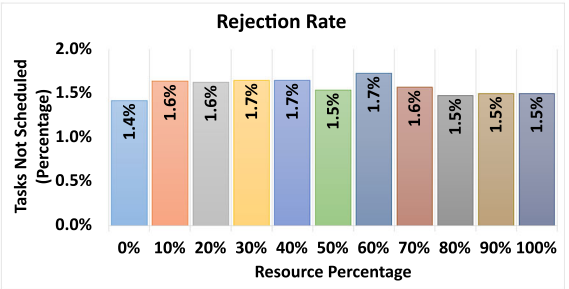
(a) Average Makespan for the percentage of VM reserved before the scheduler starts.



(b) Average Throughput for the percentage of VM reserved before the scheduler starts.

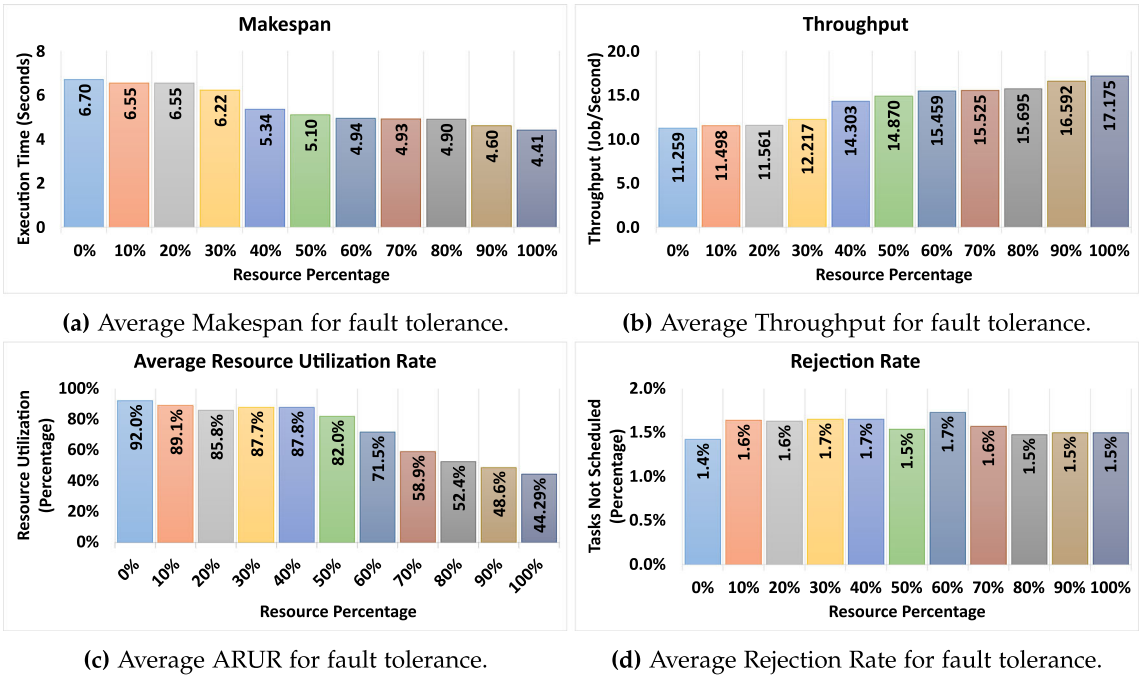


(c) Average ARUR for the percentage of VM reserved before the scheduler starts.



(d) Average Rejection Rate for the percentage of VM reserved before the scheduler starts.

Fig. 5 Results for the percentage of VM reserved before the scheduler starts on GoCJ Workload using the DFARM version scheduler



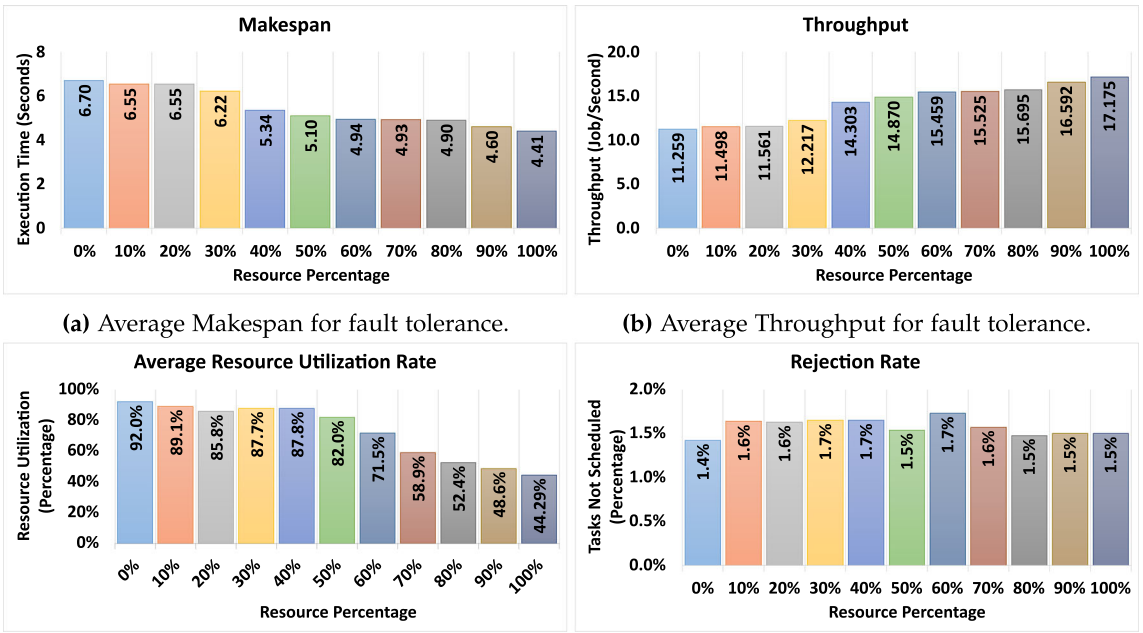
(a) Average Makespan for fault tolerance.

(b) Average Throughput for fault tolerance.

(c) Average ARUR for fault tolerance.

(d) Average Rejection Rate for fault tolerance.

Fig. 6 Results for fault tolerance on Synthetic Workload using DFARM scheduler



(a) Average Makespan for fault tolerance.

(b) Average Throughput for fault tolerance.

(c) Average ARUR for fault tolerance.

(d) Average Rejection Rate for fault tolerance.

Fig. 7 Results for fault tolerance on GoCJ Workload using DFARM version scheduler

4.4.3 Fault tolerance

Fault tolerance is the number of tasks that would either be replicated or resubmitted. Here, in this work, we represent the fault tolerance factor as a replication–resubmission ratio. Figure 6 shows the performance results for the synthetic workload when experimented with using different replication–resubmission ratios (i.e., 0–100%). Figure 7 shows performance results for the GoCJ benchmark workload when fault-tolerance mechanisms are considered based on multiple replication–resubmission ratios (i.e., 0–100%).

The makespan (i.e., the execution time for the full workload) for the synthetic dataset is depicted in Fig. 6a.

The results shown in Fig. 6b depict that replication–re-submission ratio at 0% attained 17.25%, 15.88%, 16.37%, 15.95%, 17.17% lower makespan as compared to using the replication–resubmission ratio of 20%, 40%, 60%, 80%, and 100%, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 7a. These results show that the replication–resubmission ratio when used as 0% attained 35.74%, 43.64%, 51.76%, 53.22%, 57.81% lower makespan as compared to replication–re-submission ratio of 20%, 40%, 60%, 80%, and 100%, respectively.

The throughput (i.e., number of jobs executed per second) for the synthetic dataset is depicted in Fig. 6b. The results shown in Fig. 6b depict that replication–resubmission ratio when set at 0% attained 16.38%, 15.80%, 15.93%, 15.43%, 16.44% higher throughput as compared to replication–resubmission ratio set at 20%, 40%, 60%, 80%, and 100%, respectively. The throughput results for the GoCJ benchmark dataset are depicted in Fig. 7b. These results show that replication–resubmission ratio set at 0% attained 25.76%, 32.28%, 38.69%, 41.33%, 44.64% higher throughput as compared to replication–resubmission ratio set to 20%, 40%, 60%, 80%, and 100%, respectively.

The ARUR (i.e., average resource utilization) for the synthetic dataset is depicted in Fig. 6c. The results shown in Fig. 6c depict that replication–resubmission ratio when set at 0% attained 2.65%, 0.84%, 1.52%, 3.07%, 6.31% higher ARUR as compared to replication–resubmission ratio set at 20%, 40%, 60%, 80%, and 100%, respectively. The throughput results for the GoCJ benchmark dataset are depicted in Fig. 7b. These results show that replication–resubmission ratio set at 0% attained 17.50%, 28.24%, 34.35%, 37.87%, 41.26% higher throughput as compared to replication–resubmission ratio set to 20%, 40%, 60%, 80%, and 100%, respectively.

The rejection rate (i.e., number of tasks rejected) for the synthetic dataset is depicted in Fig. 6d. The results shown in Fig. 6d depict that replication–resubmission ratio at 0% attained 17.84%, 18.49%, 18.33%, 16.20%, 17.02% lower rejection rate as compared to 20%, 40%, 60%, 80%, and 100%, respectively. The makespan results for the GoCJ benchmark dataset are depicted in Fig. 7d. These results show that replication–resubmission ratio at 0% attained 30.05%, 169.40%, 311.48%, 374.86%, 445.36% lower rejection rate as compared to 20%, 40%, 60%, 80%, and 100%, respectively.

5 Results and discussion

The proposed DFARM scheduler exhibits significant performance in terms of lowering the task rejection rate for the execution of both the synthetic and Google-like GoCJ benchmark datasets. As presented in Figs. 2 and 3, the proposed DFARM scheduler not only reduces the task rejection rates but also attains a competitive performance for makespan, throughput, and ARUR. These results are evidence of efficient scheduling not only in terms of the makespan, throughput, and ARUR along with the standout attainment of lower task-rejection rates as compared to other state-of-the-art scheduling heuristics such as RALBA.

The disparities of rejection rate and makespan between Figs. 2 and 3 highlight the impact of datasets on a deadline-constrained task. It shows that, given a small number of tasks, most of the schedulers tend to perform similarly, however in the case of a more realistic and large workload the performance differences are more clear and the proposed DFARM scheduler has a notable performance for the GoCJ benchmark dataset.

The impact of VM reservation on performance was depicted in Figs. 4 and 5. The experimental results show that the more VMs a scheduler has readily available at the start, the better it will perform in terms of performance i.e., makespan, throughput, etc. For example, the realistic and large dataset i.e., the GoCJ dataset depicted in Fig. 5a and b showcase this aspect. Moreover, Fig. 5d reveals the impact on the rejection rate which is minimal across configurations of VM allocations. From the data, it is evident that around 70% reservation rate of VM seems to be an ideal value based on the local peak appearing in Fig. 5c.

Figure 4c shows that ARUR-related performance for the synthetic dataset is not as notable as in the GoCJ dataset. The ARUR performance behavior is as expected as the synthetic workload is small and it uses only a small number of VMs as compared to the GoCJ dataset where more VMs are employed. From these results (i.e., related to ARUR performance), we can conclude that having the scheduler dynamically acquire resources as needed is the preferred solution, especially for small workloads i.e., the more freedom we allow the better the DFRAM scheduler performs.

Figure 7 showcases the effects of employing the fault-tolerance factor for the execution of the GoCJ workload. The data shows that a higher fault tolerance value leads to sub-par performance and an increased rejection rate. However, a failure in any host machine can lead to a fatal loss, because of this, the fault tolerance should be considered in a Cloud datacenter. However, the factor to include fault tolerance must be carefully studied especially when the workload contains deadline-constrained tasks too. Our experiments and the employed datasets (especially the GoCJ benchmark) show an optimal value of a fault-tolerance factor is around 30% to take with failures and provide reasonable performance.

6 Conclusions

In this paper, we proposed DFARM, a dynamic scheduler that prioritizes tasks by deadline and supports fault tolerance via a hybrid replication–resubmission mechanism, where both methods are based on the task's length relative to its deadline. The scheduler is compared with the other state-of-the-art schedulers considering the metrics such as makespan, throughput, ARUR, and rejection rate under synthetic workload and Google-like workload. The experimental data clearly shows its lead in rejection rate as well as makespan, throughput, and ARUR compared to RS, RS, MCT, RALBA, and OG-RADL. Not only is it compared with other schedulers, but multiple variants of the scheduler are also included in the comparison to showcase the effect of each component of the DFARM. Likewise, the effects of dynamic resource acquisition and fault tolerance are also present. In the future, we would extend DFARM by adding support for Softline scheduling. The approach can also be extended by considering workflow scheduling where tasks are dependent and a group of tasks must be executed in a certain order.

Acknowledgements The European Union (Horizon Europe Graph-Massivizer, 101093202) and the Austrian Research Promotion Agency (FFG Kärntner Fog, 888098) funded this work.

Author contributions Ahmad Awan: literature review, implementation, detailed solution design, experimentations. Muhammad Aleem: idea formulation, supervision, detailed solution design. Altaf Hussain: proposed architecture, draft review, experimentation validation. Radu Prodan: proposed architecture, experimental plan and design, writeup review.

Funding The European Union (Horizon Europe Graph-Massivizer, 101093202) and the Austrian Research Promotion Agency (FFG Kärntner Fog, 888098) funded this work.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Hussain, A., Aleem, M., Khan, A., Iqbal, M., Islam, A.: RALBA: a computation-aware load balancing scheduler for cloud computing. Clust. Comput. **21**, 09 (2018)
2. Hussain, A., Aleem, M., Iqbal, M., Islam, A.: SLA-RALBA: cost-efficient and resource-aware load balancing algorithm for cloud computing. J. Supercomput. **75**, 10 (2019)
3. Marahatta, A., Wang, Y., Zhang, F., Kumar, A., Sah Tyagi, S., Liu, Z.: Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers. Mob. Netw. Appl. **24**, 1–15 (2019)
4. Zhou, A., Wang, S., Cheng, B., Zheng, Z., Yang, F., Chang, R.N., Lyu, M.R., Buyya, R.: Cloud service reliability enhancement via virtual machine placement optimization. IEEE Trans. Serv. Comput. **10**(6), 902–913 (2017)
5. Saidi, K., Bardou, D.: Task scheduling and VM placement to resource allocation in cloud computing: challenges and opportunities. Clust. Comput. **26**(5), 3069–3087 (2023)
6. AbdElfattah, E., Elkawkagy, M., El-Sisi, A.: A reactive fault tolerance approach for cloud computing. In: 2017 13th International Computer Engineering Conference (ICENCO), pp. 190–194 (2017)
7. Arabnejad, V., Bubendorfer, K., Ng, B.: Scheduling deadline constrained scientific workflows on dynamically provisioned cloud resources. Future Gener. Comput. Syst. **75**, 348–364 (2017)
8. Kumar, M., Sharma, S.: Deadline constrained based dynamic load balancing algorithm with elasticity in cloud environment. Comput. Electr. Eng. **69**, 395–411 (2018)
9. Nabi, S., Ahmed, M.: OG-RADL: overall performance-based resource-aware dynamic load-balancer for deadline constrained cloud tasks. J. Supercomput. **77**, 07 (2021)
10. Garg, N., Singh, D., Singh Goraya, M.: Deadline aware energy-efficient task scheduling model for a virtualized server. SN Comput. Sci. **2**(3), 169 (2021). https://doi.org/10.1007/s42979-021-00571-2
11. Chinnathambi, S., Santhanam, A., Rajarathinam, J., Maruthamuthu, S.: Scheduling and checkpointing optimization algorithm for byzantine fault tolerance in cloud clusters. Clust. Comput. **22**, 11 (2019)
12. Adhikari, M., Amgoth, T.: Heuristic-based load-balancing algorithm for IaaS cloud. Future Gener. Comput. Syst. **81**, 156–165 (2018)
13. Iftikhar, S., Ahmad, M.M.M., Tuli, S., Chowdhury, D., Xu, M., Gill, S.S., Uhlig, S.: HunterPlus: AI based energy-efficient task scheduling for cloud-fog computing environments. Internet Things **21**, 100667 (2023)
14. Nazeri, M., Khorsand, R.: Energy aware resource provisioning for multi-criteria scheduling in cloud computing. Cybern. Syst. (2022). https://doi.org/10.1080/01969722.2022.2071409
15. Alaei, M., Khorsand, R., Ramezanpour, M.: An adaptive fault detector strategy for scientific workflow scheduling based on improved differential evolution algorithm in cloud. Appl. Soft Comput. **99**, 106895 (2021)
16. Hussain, A., Aleem, M.: GoCJ: Google cloud jobs dataset for distributed and cloud computing infrastructures. Data **3**(4), 38 (2018)
17. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **61**(6), 810–837 (2001)
18. Zhu, X., Yang, L.T., Chen, H., Wang, J., Yin, S., Liu, X.: Real-time tasks oriented energy-aware scheduling in virtualized clouds. IEEE Trans. Cloud Comput. **2**(2), 168–180 (2014)
19. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw.: Pract. Exp. **41**(1), 23–50 (2011). https://doi.org/10.1002/spe.995

Ahmad Awan is a Masters's student in computer science at the National University of Computer and Emerging Sciences, Islamabad Pakistan. His research interests are cloud computing and scheduling issues in distributed systems.

Muhammad Aleem received a Ph.D. degree in computer science from the Leopold-Franzens- University, Innsbruck,Austria in 2012. His research interests include parallel and distributed computing comprising programmingenvironments, multi-/many-core computing, performance analysis, cloud computing, and big-data processing. He iscurrently working as a Professor at National University of Computer and Emerging Sciences, Islamabad, Pakistan.He has co-authored over 70 research publications and 2 books.

Altaf Hussain is working as an Associate Professor and Head of the Computer Science department at the Institute of Space Technology (KICSIT Campus), Islamabad, Pakistan. He received his Ph.D. in Computer Science from the Capital University of Science & Technology, Islamabad, Pakistan in 2020. His research interests include parallel and distributed computing comprising programming environments, performance analysis, cloud computing, data mining, and big-data processing. He is serving as a reviewer for many reputed computer science journals. He has more than 15 years of teaching, research, and development experience.

Radu Prodan is a professor in distributed systems at the Institute of Information Technology, University of Klagenfurt, Austria. Previously, he was an associate professor at the University of Innsbruck, Austria. He received his Ph.D. in 2004 from the Vienna University of Technology. His research interests are performance, optimization, and resource management tools for parallel and distributed systems. He participated in numerous projects and coordinated the European Union projects ARTICONF and Graph-Massivizer. He co-authored over 300 publications and received three IEEE best paper awards.