Preprint of paper to appear in Concurrency - Practice and Experience

An Efficient Deadline Constrained and Data Locality Aware Dynamic Scheduling Framework for Multi-Tenancy Clouds

Jia Ru*[1] | Yun Yang[1] | John Grundy[2] | Jacky Keung[3] | Li Hao[4]

[1]School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia
[2]Faculty of Information Technology, Monash University, Melbourne, Australia
[3]Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China
[4]SoptAI Co.Ltd, Singapore

Correspondence
Jia Ru, School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia. Email: jiaruweiwei@gmail.com

Present Address
Swinburne University of Technology, PO Box 218, Hawthorn, Melbourne, Australia, 3122

Summary

Scheduling and resource allocation in clouds is used to harness the power of the underlying resource pool. Service providers can meet QoS requirements of tenants specified in SLAs. Improving resource allocation ensures that all tenants will receive fairer access to system resources, which improves overall utilisation and throughput. Real-time applications and services require critical deadlines in order to guarantee QoS. A growing number of data-intensive applications drive the optimisation of scheduling through utilising data locality in which the scheduler locates a task and ensures the task's relevant data to be on the same server. Choosing suitable scheduling mechanisms for running applications that support multi-tenancy has consistently been a major challenge. This work proposes a new adaptive Deadline constrained and Data locality aware Dynamic Scheduling Framework – 3DSF – that orchestrates different schedulers based on varied requirements. This framework considers tenants' deadline-based QoS requirements, cloud system's performance and a method of resource allocation to improve resource utilisation, system throughput and reduce jobs' completion time. 3DSF contains: (1) a real-time, preemptive, deadline constrained job scheduler, (2) an optimised data locality aware scheduler, (3) an improved Dominant Resource Fairness greedy resource allocation approach and (4) an adaptive suite to integrate above-mentioned schedulers together.

KEYWORDS:
scheduling framework, deadline, data locality, resource allocation, multi-tenancy

1 | INTRODUCTION

Cloud computing enables resources to be shared among different customers, provides computation, storage, and communications resources and also provides a variety of QoS sensitive services to different tenants. Multi-tenancy is a vital characteristic in cloud computing, as it supports scalability and birngs economic benefits to the users and service providers by sharing the same instance, cloud platform and underlying infrastructure with the isolation of shared network and computing resources. Diverse cloud computing frameworks have been developed to support scheduling in multi-tenancy clouds. Prominent examples include YARN[1], Mesos[2], MapReduce[3], Spark[4], Quincy[5] and Omega[6]. A cluster typically hosts a number of data-intensive and computation-intensive applications, such as data mining, web crawling, and network traffic analysis. These applications generate widely varied workloads, share resources, and are executed on machines with different hardware parameters. A number of frameworks, such as Hadoop[7] and Dryad[8], employ a fine-grained resource sharing model, where nodes are composed of

"slots" and jobs are divided into tasks that can be matched to slots. However, the slots with fixed resources are fixed on the nodes, which result in the mismatching of tasks' resource requirements. This reduces the ability of dynamic resource allocation. Another drawback is that the mismatch between allocated fixed resources and other related jobs, and preventing the system from achieving high resource utilisation. As each job will hardly have any chance to obtain the approximate resources required for the application to run, with the required resources running on nodes that store input data. A better approach could use Linux containers, instead of slots in order to achieve dynamic resource allocation.

Multi-tenancy allows different users to customise the currently running instances on a given machine, allowing for the customisation of instances within the same application. The shortcomings of multi-tenancy lie in its inability to effectively utilise the resources of a given system[9,10]. A tenant may have one user or multiple users, and submit their jobs with several QoS requirements. However, the demands of users are often varied, making it difficult to comply with all the users' expectations. Multi-tenancy makes scheduling issues on clouds more complicated. For real-time applications and services, compliance with deadlines is a major criterion in judging the QoS. Precisely predicting service performance according to statistics analysis of the system and user information helps us to guarantee system's QoS, reduce the Service Level Agreement (SLA) violations and avoid the over/under provision of system resources[11,12]. The primary challenge for applications is to meet SLAs on the comple-tion of their jobs and before their deadlines[13]. For example, financial applications often have strict job deadlines and are often more willing to pay more for compliance with the deadlines, but some scientific jobs prefer to trading a bounded delay for lower costs. Similarly, completion time is critical for web searches, as only a small fraction of delayed results can result in an impor-tant loss in revenue through missed purchases, reductions in search queries or reductions in advertisement click-through rates[13]. As a result, scheduling mechanisms should differentiate jobs according to their importance or priority[13]. Deadline constrained jobs are thus generally more important than regular jobs.

Multi-tenancy scheduling enables multiple instances of an application to occupy and share resources from a large pool, allow-ing different users to have their own version of the same application running and coexisting on the same hardware. Moreover, to support the proposed traditional multi-tenancy frameworks, researchers have also proposed some algorithms to meet a frame-work's functional requirements[14,15]. For instance, a framework proposed in[14] periodically re-allocates resources to tenants, aiming to maximise the resource utilisation while tolerating a low risk of SLA violations, especially for highly dynamic work-load. In this framework[14], a dynamic resource allocation algorithm for DBaaS is proposed if there is a higher variance intensity. Additionally, an extensible dynamic provisioning framework presented in[15] begins with a Tenancy Requirements Model (TRM). The model in[15] is based on the mapping of functional and non-functional tenancy requirements with appropriate resources, their parameters, and the health monitoring policy, which allows dynamically re-provisioning for existing tenants based on either changing tenancy requirements or health grading predictions. Many other multi-tenancy cloud computing frameworks have been proposed to support job/task scheduling or resource allocation, such as YARN[1], AWS cloud adoption framework (AWS CAF)[16], Patent[17], Omega[6], Quincy[5], Canary[18], QRSF[19], ERA[20], and works[21,22]. However, none of these frameworks focus on job scheduling, task scheduling and resource allocation concurrently, while also ensuring that they are complying with QoS regulations and implementing efficient data locality. Most of these multi-tenancy scheduling frameworks[6,5,1] do not consider the importance of deadline constraints to the users. Moreover, even if some frameworks do consider the deadline issue, they use very simple prediction models in order to predict their service time and requirements. However, imprecise prediction models would degrade the performance of deadline constrained schedulers. Data locality is one of the most important considerations for multi-tenancy scheduling in order to improve the response time and throughput. Enhancing data locality is very important to minimise data shuffling across nodes and to reduce a job's sojourn time. Some works[23,2] focus on data locality issue. To our best knowledge, these works only emphasise data locality without considering other parameters.

Network bandwidth is a precious resource in a cloud cluster and maintaining high bandwidth between arbitrary pairs of computers becomes increasingly expensive with the growth of cloud cluster size, particularly since hierarchical networks are the norm for current distributed computing clusters[3,8]. The communication between machines in the same rack is *"cheaper"* than communication between racks[5]. If computations are not located close to their input data, the network has the potential to be a bottleneck. Decreasing network traffic can simplify capacity planning. If parallel jobs are within high cross-cluster network traffic, they will compete for bandwidth, and modeling this dynamic network congestion strongly will complicate performance prediction[5]. Because of these reasons, optimising the placement of computations to minimise network traffic is one of preliminary goals in a data-intensive computing platform, especially when the applications require big data[5].

As for the time-variance of workloads[21], it is desired to implement cloud resource allocation in a dynamic and adaptive manner. This implementation of cloud resources can reduce the amount of hardware required for a workload[6,24]. A scheduling framework requires an efficient resource allocation strategy to exhibit high dimensions in resource utility, which can improve the

efficiency of resource utilisation, reduce tradeoffs and job completion time[21]. A core feature of cloud systems is that resources such as CPU, memory and I/O can be provisioned to the applications based on their requirements. The jobs are classified into CPU-intensive, memory-intensive and I/O-intensive based on their dominant resources requirements and consumptions[9]. In particular, when CPU-intensive, memory-intensive and I/O-intensive jobs are running simultaneously, the main problems then become how to properly schedule deadline-oriented jobs with precedence over other jobs and how to assign different jobs to different resources by balancing the tradeoff between data locality and run time blocking.

The challenge for scheduling strategies is that the fairness of resource allocation often collides with data locality and high deadline-based QoS. Dispatching jobs to the machines which have required data is *"cheaper"* than allocating jobs to other machines. To optimise data locality, the scheduling strategy can delay a job until there are enough available resources that the job requires. However, fairness benefits from allocating the best available resources to a job as soon as possible after they are requested, even if they are not the resources closest to the data[5]. Meanwhile, the jobs should also be guaranteed to finish before their deadlines. The goal of cloud providers is to try to finish and satisfy all the tenants' requests and facilitate the cloud system's resource utilisation, while the goal of tenants is to gain high quality service and finish their jobs as soon as possible for as little money as possible. In order to meet tenants' deadline-based QoS requirements, while also enhancing data locality and fairer allocation of provisioning resources, we propose a highly efficient ***deadline*** constrained and ***data*** locality aware ***dynamic*** scheduling framework, named ***3DSF***.

Our proposed scheduling framework – 3DSF that integrates different schedulers together is essential to realise all these goals, simultaneously. The framework is motivated by real-world applications, especially for data-intensive applications, which process and generate a large volume of data. Our novel research, from both cloud providers' and tenants' perspectives, investigates key issues on how to provide efficient scheduling policies while also meeting QoS requirements, improving system's throughput and resource utilisation, achieving better data locality, and reducing job completion time and network costs. We use a mechanism to determine when moving computation (putting jobs to the nodes with sufficient resources) is optimal and when moving data (putting jobs to the nodes with required data) is optimal and when to reasonably schedule jobs based on their deadlines in order to achieve a balance between resource fairness, data locality and the maintenance of a higher QoS, simultaneously.

Our ***3DSF*** uses an adaptive suite to choose the corresponding scheduling strategy that satisfies the differing profiles of jobs that are submitted with factors such as the specific resource requirements, customised data locality and other special QoS. This ***3DSF*** focuses on *job*, *task* and *bottom hardware* levels and does not consider the interference of multi-tenants. Our work does not consider the privacy and security of multi-tenants and assume that jobs from multi-tenants are independent. In addition, our work also does not concern the relevance of data, which are the possible shortcomings of our work. We develop and implement ***3DSF*** scheduling framework in the open source Hadoop YARN implementation and perform comprehensive evaluations with various MapReduce workloads. Currently, our ***3DSF*** mainly works on MapReduce workloads regardless of application types. The probability of missing deadline in Hadoop workloads can be minimised by exploiting the dynamics in resource availability and the flexibility in job scheduling. We lay the foundation for achieving these objectives with the key contributions of:

1. To satisfy each tenant's QoS and reduce SLA violations, a real-time, preemptive, deadline constrained scheduler using queuing theory – PDSonQueue – is proposed, which enables jobs to better meet their QoS and also shortens a job's completion time and improves the system's throughput and performance;

2. To facilitate task scheduling and find the relationships between tasks and data easier, an optimised data locality aware scheduler for balancing time consumption and network bandwidth traffic – DLAforBT – is proposed, which improves data locality for tasks, with the optimal placement policy exhibiting a threshold-based structure, maximises resource utilisation, improves throughput and shortens completion time of jobs;

3. To dynamically provision resources and enhance fairer resource sharing, a fairer Dominant Resource Fairness (DRF) resource allocation mechanism with a 3-dimensional demand vector <CPU, memory, disk I/O> is presented to support disk I/O resources as the third dominant shared resource while maximising the number of jobs being allocated in the cluster to assist dynamic resource allocation; and

4. To efficiently run different applications for tenants, a scheduling framework is proposed, which comprises each of the above individual plug-in schedulers used in tandem. Regarding to the diversity of tenants' resources and QoS requirements and heterogeneity of cloud nodes, our scheduling framework uses an auto-adaptive suite to choose the correspond-ing scheduler or appropriate combinations according to different scenarios and handle all the aforementioned issues simultaneously.

The rest of the paper is organised as follows. Section 2 introduces the motivation of this work. Section 3 discusses the related work. Section 4 presents the overall design of our deadline constrained and data locality aware dynamic scheduling framework – 3DSF. Section 5 presents our improved resource allocation strategy. Section 6 describes our novel data locality aware task scheduler. Section 7 discusses our efficient deadline constrained job scheduler. Section 8 describes the integration of individual schedulers and the detailed architecture of our scheduling framework – 3DSF. Section 9 evaluates our scheduling framework by conducting a series of experiments and presents observations and results with threats to validity addressed in Section 10. Section 11 concludes the work and outlines future work.

2 | MOTIVATION

Driven by large Internet services and a growing number of data-intensive scientific applications, researchers and practitioners have been developing diverse cloud computing frameworks to support scheduling in the cluster. Unfortunately, these frameworks do not resolve scheduling and resource allocation issues directly and cannot achieve high utilisation, efficient resource sharing, impactful data locality or satisfy users QoS at the same time. The main obstacle to this is the mismatch between the allocation granularities of these solutions for existing frameworks. There are studies on dynamic resource allocation and the impact that performance isolation has between applications and tenants in a shared computing environment possibly highlighting a new challenge for cloud computing[21,22,23]. Both over- and under-allocation of resources are undesirable and will adversely impact the tenant itself and others in the shared cloud environment. Furthermore, data locality is essential for reducing total job execution time, improving cross-rack communication and improving total throughput[25]. The other challenge for scheduling strategies is that the fairness of resource allocation often collides with data locality and deadline-based QoS. Finding a way to determine when moving computation (putting jobs to the nodes with sufficient resources) is optimal and when moving data (putting jobs to the nodes with required data) is optimal and when to reasonably schedule jobs based on their deadlines in order to achieve a balance between resource fairness, data locality and the maintenance of a higher QoS. Solving the conflict between satisfying deadline-based QoS requirement, fairer resource allocation and data locality simultaneously is a crucial issue. It is a better way using individual approach to solve each of the challenging issues and integrate all the approaches together as an entity. Therefore, proposing a highly efficient deadline constrained and data locality aware dynamic scheduling framework is essential.

We take a motivating example of our work as a scenario for problem analysis and point out the key challenges that need to be tackled. Assume that tenant ***A*** submits a CPU-intensive application (job) ***a*** with a deadline, tenant ***B*** submits a data-intensive application (job) ***b*** which consumes more memory resource and has a deadline, and tenant ***C*** submits a data-intensive application (job) ***c*** which requires more I/O resource. To improve the system's utilisation, it is better to dynamically allocate resources to tenants ***A***, ***B*** and ***C*** according to the current system's remaining available resources. When greater amounts of CPU-resources are available in the system, the job of tenant ***A*** should be prioritised to gain higher system's utilisation, as job ***a*** is a CPU-intensive application. How many jobs of ***a***, ***b***, ***c*** should be allocated with resources to run in the system is one consideration. Meanwhile, both applications ***a*** and ***b*** have deadline requirements. To guarantee that tenant ***A***'s and tenant ***B***'s jobs complete before their deadlines, applications ***a*** and ***b*** should have higher priorities than application ***c***. However, applications ***b*** and ***c*** are data-intensive, meaning that these applications will read and write large amounts of data during execution. Scheduling jobs on a server which has sufficient resources and relevant data can reduce completion time and network bandwidth I/O consumption.

If job ***b***'s execution time can be shortened, job ***b*** has higher probability to be accomplished by its deadline. To achieve all these goals, one single scheduler is hard to realise all the functions. Thus, a scheduling framework that integrates different schedulers together is essential to realise all these goals. This framework needs to consider the cloud system with various node performance and running jobs with different resources requirements. Intuitively, the proposed scheduling framework should schedule jobs and dynamically allocate resources with regard to users' QoS, data locality and network bandwidth utilisation, concurrently, while improving the system's throughput, reducing deadline-based SLA violation and completion time, minimising the system's overhead, and maximising resource utilisation.

3 | RELATED WORK

Multi-tenancy, as a new software architectural pattern, enables multiple instances of an application to occupy and share computing resources[9,10]. Tenants need a scheduling framework that can efficiently run their applications in the cloud. Suitable scheduling mechanisms for running such applications for multi-tenants in the cloud have been a major challenge, due to the

problem of finding the best match of resource-workload pair and meeting QoS requirements simultaneously. The size of cloud data is expanding at an impressive speed, and a large amount of data is generated and executed by cloud applications with data-intensive characteristics. This brings to researchers a new challenge of deploying computation and data intensive applications without any infrastructure investment.

3.1 | Resource Provisioning

Resource allocation is a crucial issue in cloud computing, especially for on-demand resource offering. Multiples studies have been conducted. Dynamic Resource Allocation (DRA) strategies can be categorised with amounts of taxonomies, in terms of optimisation criteria (performance or energy), target architecture (homogeneous or heterogeneous), or criticality (hard or soft real-time)[26,27]. The works in[27,28,29,30] all focus on dynamic resource allocation without considering the disk I/O resources. However, disk I/O is an extremely important factor when seen through its impact on traffic. The work in[28] proposes the Dynamic Priority (DP) parallel task scheduler for Hadoop. It makes users to control their allocated capacity through adjusting their spending over time. The work in[30] proposes a two-tiered on-demand resource allocation mechanism consisting of both a local and global resource allocation to provide on-demand capacities to the concurrent applications based on control theory. The max-min fairness algorithm[31] is one of the most popular resource allocation mechanisms currently being used. This was originally proposed for computer networks to facilitate better scheduling capability. Currently, it is being widely used in cloud computing, with different versions of implementations such as Fair Scheduler[32], Capacity Scheduler[33], Choosy[31], Quincy[5], Carbyne[34] and DRF[35]. These schedulers attempt to optimise the minimum resource allocation needed received by each prospective tenant[35].

3.2 | Deadline Constraint

The predictability of jobs' service time helps deadline constrained jobs to be finished within deadlines and to guarantee jobs' QoS. A precise performance prediction model for services including job's execution time and waiting time is needed, based on systematic statistical analysis and history results by using existing performance estimation techniques, e.g. analytical modelling, queuing modelling, task modelling, and empirical and historical data[36,37,38]. Queuing theory is a collection of mathematical models of distinct queuing systems to systematically study waiting time and queues. Queuing models are widely used to model service performance in cloud computing, aiming to optimise energy consumption, resource allocation, performance prediction, resource management, load sharing, etc.[39,40,11]. For instance, work in[11] uses an M/M/C/C queuing system with different priority classes to model cloud datacenters, in order to support decision making with respect to resource allocation when different clients negotiate different SLAs. Earliest Deadline First (EDF) is a dynamic priority real-time scheduling algorithm which pay regard to time constraints of a task in scheduling them for execution[41]. Work in[42] proposes a queuing theory based performance model for a multi-priority preemptive M/G/1./EDF system. Their presented model estimates the mean waiting time for a given class according to the higher and lower priority tasks receiving service prior to the target and the mean residual service time experienced. Additional time caused by preemptions is estimated as part of mean request completion time. Works in[43,44] propose RDS, a Resource and Deadline-aware Hadoop job Scheduler that takes future resource availability into consideration when minimising job deadline misses. They formulate the job scheduling problem as an online optimisation problem and solve it using an efficient receding horizon control algorithm. Work in[45] proposes and develops BIG-C, a container-based resource management framework for Big Data cluster computing. Work in[46] proposes a preemptive deadline job scheduling algorithm based on the current status of the system and the job execution cost model to obtain the sub-optimal minimal completion time within jobs' deadlines. The key design is to leverage lightweight virtualisation, a.k.a, containers to make tasks preemptable in cluster scheduling. Stratus[47] proposes a cost-aware container scheduler, which orchestrates batch job execution on virtual clusters, dynamically allocated collections of virtual machine instances. To our best knowledge, the preemption performance in current preemptive schedulers such as[42,46,48,49] is limited and most cloud deadline schedulers rarely consider preemption.

3.3 | Data Locality Awareness

Data locality is one of critical factors affecting a cloud system's performance. A large number of researches[5,50] have been carried out to address this challenge. BGMRS - a bipartite graph based MapReduce Scheduler[36] takes data locality into account, in terms of computing resource allocation for shortening the data access time of a job. BGMRS transforms deadline-constrained

scheduling problem into a graph problem by minimum weighted bipartite matching. However the prediction model of Map/Reduce execution time is very simple and inaccuracy. However, some works in [51,52,53] use directed acyclic graph (DAG) to model parallel jobs, where each node of the DAG is a sequential sequence of instructions and each edge is a dependence between nodes. BAR, a data locality task scheduler [54] schedules tasks by taking a global view and adjusts task data locality dynamically based on network state and workload in the cloud cluster. However, this work uses flow network, rather than more popular topology network to model data placement. Work in [55] proposes BOLAS, a task scheduling algorithm, which models the scheduling process as a bipartite-graph matching problem to assign data block to the nearest task. BOLAS solves the model using Kuhn-Munkres optimal matching algorithm. However, the weakness of BOLAS is to assume all the nodes are homogeneous, which is unrealistic. Work in [56] proposes a Hadoop Constraint Programming based Resource Management algorithm (HCP-RM) that incorporates a technique for handling data locality. However, this work solves the scheduling problem on a closed system with a fixed number of MapReduce jobs. Work in [57] studies the data locality problem by utilising data migration and hotspot file prediction for lowering task execution waiting delay, in which hot files are transferred periodically to multiple data centers during information interaction. A key question is how to schedule tasks in the vicinity of their inputs so as to diminish shuffled data, reduce unnecessary data transfer and network traffic, and improve system's performance [58]. To make scheduling decisions, constructing a performance estimation model that predicts the execution time and waiting time of jobs in a cloud system is essential. Work in [59] proposes a novel data distribution model which distributes data based on the node's capacity level respectively, in which the node classification algorithm can improve data locality. Work in [60] introduces a novel one-to-one sampling method to calculate average execution time of map/reduce tasks respectively. However this kind of simple sampling method would reduce the accuracy of execution time estimation.

3.4 | Scheduling Framework

Many cloud computing frameworks [61,62,23] have been proposed to support job/task scheduling or resource allocation. YARN takes the cluster resource management capabilities from the MapReduce [3] system, and then uses these new engines to utilise the generic cluster resource management capabilities. YARN introduces a *container* concept to realise dynamic resource allocation. Omega [6] attempts to lean more heavily towards distributed, multi-level scheduling. This system reflects a greater focus on scalability but makes it harder to enforce global properties such as capacity/fairness/deadlines. The work in [63] proposes a novel predictive scheduling framework to enable fast and distributed stream data processing, which features topology aware performance prediction and predictive scheduling. However, none of these frameworks focus on job scheduling, task scheduling and resource allocation while also ensuring that they are complying with QoS regulations and implementing efficient data locality. Most scheduling frameworks [1,5,6] do not consider the importance of deadline constraints to the users. However, some deadline constrained frameworks [60,23] only use very simple prediction models to predict their service times and requirements. Imprecise prediction models would reduce the performance of deadline constrained schedulers. Data locality is one of the most important considerations for task scheduling in order to improve the response time and throughput. However, for reasons like data skew and slots constraints, the goal of trying to place the computation task closest to the data location is not easy to achieve. To our best knowledge, none of these cloud scheduling frameworks consider "job" level and "task" level scheduling problems concurrently.

4 | A DEADLINE-CONSTRAINED AND DATA LOCALITY-AWARE DYNAMIC SCHEDULING FRAMEWORK

Resource management and scheduling in multi-tenancy cloud computing becomes one of the most sophisticated missions due to the inherent heterogeneity and resource isolation [22]. The purpose of our proposed scheduling framework is to maximise resource utilisation, reduce the deadline-based SLA violation, improve system's throughput, data locality, and network bandwidth utilisation, reduce completion time of jobs and to minimise a system's overhead. As shown in Figure 1, this scheduling framework operates through layers, ranging from the software layer to the infrastructure layer, which comprises some components: a deadline constrained preemptive job scheduler on the top layer, an optional data locality aware task scheduler on the middle layer, a resource allocation strategy on the bottom layer and an adaptive suite to integrate mentioned above schedulers together. ***On the top layer***, users or tenants submit jobs (application instances) to the cloud with different QoS requirements. Considering the diversity of QoS, "deadline" and "priority" are our selected QoS factors. Regarding the features of our scheduling framework's QoS, there are deadline jobs and non-deadline jobs. Additionally, the scheduling framework assigns the priority to jobs. From

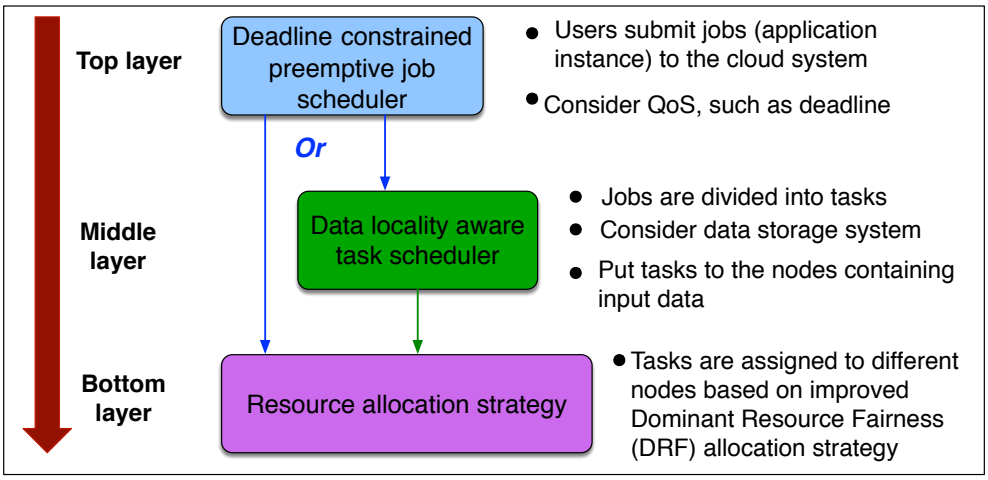

FIGURE 1 Design of our scheduling framework

the applications' (users') perspective, some jobs have higher priority than others, and the jobs with deadline are more important than others. Higher priority jobs can preempt resources from lower priority jobs. When there are insufficient resources, the scheduling framework could suspend low priority jobs and allow deadline jobs to continue to run by preempting resources in order to ensure that they complete prior to their deadline. ***On the middle layer***, according to popular cloud data processing model, the submitted jobs (application instances) are divided into tasks first, and the tasks are assigned to different cloud nodes, since cloud computing processes data in parallel. If jobs are data or computation-intensive and require a large amount of data to process the tasks, a data locality aware task scheduler is invoked. Intuitively, putting a task near its required data block will reduce network traffic and cost. Otherwise, extra costs will incur to transfer the data. Finding the relationship between data blocks and tasks and putting tasks on the "right" nodes is an effective way to solve data locality issue. On the other hand, when the jobs are smaller, after jobs partition, the tasks can skip this optional data locality aware task scheduler and directly go to the bottom layer to gain the resource allocation. ***On the bottom layer***, the allocation strategy determines which job (user) should run next according to current system's resource utilisation. It is worth noting that resource allocation strategies typically do not consider how they will deploy the related sub tasks to the cloud nodes and also do not consider jobs' QoS. Our resource allocation policy considers the whole system's utilisation and determines how many jobs and what kinds of jobs are to be processed.

The overall resource allocation strategy used in our scheduling framework was proposed in our earlier work[24], and the described deadline constrained scheduler and data locality aware scheduler in ***3DSF*** have been extended from our earlier work[64],[65]. A major new contribution of this work was to integrate all these three schedulers together, with highly efficient performance. We have also set up an adaptive suite to automatically invoke schedulers according to different tenant's requests.

5 | OUR ENHANCED FAIRER RESOURCE ALLOCATION STRATEGY

Cloud computing enables resources to be shared between different tenants. The inherent challenge of resource allocation comes when different tenants may have diverse resource requirements (CPU, memory and I/O). In order to provide better resource allocation, the Dominant Resource Fairness (DRF) approach[35] has been developed to address the "fair resource allocation problem" that occurs at the application layer for multi-tenant cloud applications. Nevertheless, conventional DRF only considers the interplay of CPU and memory, regardless of disk I/O resources and network bandwidth, which may imbalance disk I/O utilisation, even allow some tenants to occupy an unbalanced proportion of disk I/O resources and result in over allocation of resources to one tenant's application to the detriment of others. We propose an improved Dominant Resource Fairness (DRF) algorithm with a 3-dimensional demand vector <CPU, memory, disk I/O> to support disk I/O resources as the third dominant shared resource[24]. Our technique is integrated with Linux *Cgroup* to control resource utilisation and incorporates data isolation

to avoid undesirable interactions between co-located tasks. Our algorithm ensures all tenants receive system resources fairly, which improves the overall utilisation and throughput of a system [†].

5.1 | Dominant Resource Fairness (DRF)

DRF[35] is a method to provide fair allocation for heterogeneous resources. DRF attempts to maximise the "minimum dominant share" of resources for all users. DRF attempts to fairly distribute memory and CPU resources among these different types of jobs in a mixed-workload cluster[66]. But it does not consider disk I/O resources and network bandwidth, which is its main weakness. Tasks consume different resources simultaneously, which may be bottlenecked and blocked on these different resources. Without controlling the disk I/O, some tasks will exhaust disk I/O, which would incur disk blocking of other tasks and increase their completion time. Improved job completion time after implementing a disk optimisation approach represents a best-case scenario[67]. DRF uses the concept of a "dominant resource" to compare multi-dimensional resources. In a multi-resource environment, resource allocation should be determined by the dominant share of an entity (user or queue), which is the maximum share that the entity has been allocated of any resource (memory or CPU). In a nutshell, DRF tries to maximise the minimum dominant share across all tasks using shared resources[66]. For instance, when user *a* runs CPU-intensive jobs (e.g. Storm-on-YARN[1]) and user *b* runs memory-intensive jobs (e.g. MapReduce[3]), DRF seeks to equalise the CPU share of *a* with the memory share of *b*. Eventually, DRF will allocate more CPU and less memory to *a*, and allocate less CPU and more memory to *b*. In a homogeneous resource environment, all the tasks require the same type of resources and hence DRF reduces to max-min fairness for the resource[35,66]. DRF algorithm[35] identifies some significant allocation properties, such as sharing incentive, strategy-proofness, envy-freeness and more importantly Pareto efficiency. The strength of DRF lies in these properties which are satisfied, especially when these properties are trivially satisfied by max-min fairness for a single resource, but are important in multiple resources. DRF provides incentives for users to share resources by guaranteeing that no user is better off in a system in which resources are statically and equally partitioned among users. DRF ensures that users do not gain a better allocation by lying about their resource demands. Moreover, DRF allocates all available resources subject to satisfying the other properties, and without preempting existing allocations. DRF ensures that no user prefers the allocation of another user[35].

5.2 | Mathematics Principles of Our Approach

Assume a cloud system with 24 virtual CPUs (vCPUs), 36 GB RAM, 54 virtual disk I/O (disk I/O), a resource sharing degree of $Lev = 1$, and 3 tenants (users). Our algorithm introduces a "resource sharing degree" concept, in which the lower the degree, the more dominant the resource for a task. Tenant *a* runs tasks with resource requirement 3-dimensional demand vector <2 vCPU, 4 GB, 3 disk I/O>, Tenant *b* runs tasks with requirement <3 vCPU, 2 GB, 6 disk I/O>, and Tenant *c* runs tasks with requirement <1 vCPU, 3 GB, 6 disk I/O>.

Naïve DRF algorithm only considers CPU and memory. In our enhanced fairer DRF allocation algorithm, CPU, memory and disk resources are all considered. The restriction of I/O speed and amount of shared storage I/O can be set for each task, job, tenant or group of tenants. $Lev = 1$ indicates the most dominant resource value. Therefore, after calculating the fraction of required resources to the total resource, the largest one is selected as the dominant share. Tenant *a* requires $\{\frac{1}{12}$ vCPU, $\frac{1}{9}$ memory, $\frac{1}{18}$ disk I/O$\}$, and its dominant share is memory. Tenant *b* needs $\{\frac{1}{8}$ vCPU, $\frac{1}{18}$ memory, $\frac{1}{9}$ disk I/O$\}$, and its dominant share is CPU. Tenant *c* requires $\{\frac{1}{24}$ vCPU, $\frac{1}{12}$ memory, $\frac{1}{9}$ disk I/O$\}$, and its dominant share is disk.

This allocation can be calculated and simplified mathematically as follows. Given x, y and z respectively stand for the number of tasks allocated by enhanced fairer DRF to Tenants *a*, *b* and *c*. Tenant *a* is allocated $<2x$ vCPU, $4x$ GB, $3x$ disk I/O >, Tenant *b* is allocated $<3y$ vCPU, $2y$ GB, $6y$ disk I/O >, and Tenant *c* is allocated $<1z$ vCPU, $3z$ GB, $6z$ disk I/O >. The total amount of resources assigned to these 3 tenants are $(2x + 3y + 1z)$ CPUs, $(4x + 2y + 3z)$ GB, and $(3x + 6y + 6z)$ disk I/Os.

Our enhanced fairer DRF algorithm attempts to equalise the dominant share of Tenants *a*, *b*, and *c*: $\frac{4x}{36} = \frac{3y}{24} = \frac{6z}{54}$. We should point out that DRF does not always equalise tenants' dominant share, since as one tenant's resource requirement is satisfied, the extra resources will be split to other tenants. If one type of resource is exhausted, the tenants that do not need that type of resource will still continue receiving higher shares of other types of resources[35].

[†]More details of this allocation strategy is described in our previous work[24]

Equation 1 provides an answer to this problem:

$$\begin{aligned} &\max(x, y, z) \qquad (maximise\ allocations)\\ &\text{constraint to}\\ &\begin{cases} 2x + 3y + z \le 24 \quad (CPU\ constraint)\\ 4x + 2y + 3z \le 36 \quad (memory\ constraint)\\ 3x + 6y + 6z \le 54 \quad (disk\ I/O\ constraint)\\ \frac{4x}{36} = \frac{3y}{24} = \frac{6z}{54} \quad (equalise\ dominant\ share) \end{cases} \end{aligned} \tag{1}$$

Solving this equation, we get $x = 4$, $y = 3$, and $z = 4$ (note that one task must be processed as an entity, so the values of x, y and z must be integers). Consequentially, Tenant *a* receives <8 vCPU, 16 GB, 12 disk I/O>, Tenant *b* receives <9 vCPU, 6 GB, 18 disk I/O>, and Tenant *c* receives <4 vCPU, 12 GB, 24 disk I/O>. Table 1 outlines how this could be achieved. Tenant *a* receives 44.44% of the memory resource; Tenant *b* receives 37.50% CPU resource; and Tenant *c* receives 44.44% of the disk resource. The system resource utilisation is considered as very high <87.50% CPU, 94.44% memory, 100.00% disk>.

In this scenario, the naïve DRF algorithm does not consider the disk I/O utilisation. Tenant *a*'s dominant share is memory ($\frac{1}{9}$), *b*'s share is CPU ($\frac{1}{8}$), and *c* share is memory ($\frac{1}{12}$). Based on the max-min principle, *c*'s task is being executed first, and then *a*'s task will be processed secondly. *b*'s dominant share ratio is the largest and runs next. After several iterations, all the disk I/O resource is exhausted. Finally, *a b* and *c* have 3, 3, 4 tasks executed, respectively. The system utilisation is <79% CPU, 83% memory, 100% disk>, which is 10% lower than our modified DRF method. Use of the naïve DRF algorithm easily results in I/O exhaustion, which unfortunately, blocks other resource usage. Our method solves this issue and can also control each kind of resource's usage to avoid some tenants occupying too many resources, which leads to reducing others' performance.

Schedule	Tenant a		Tenant b		Tenant c		CPU total allocation	Memory total allocation	Disk I/O total allocation
	resource shares	dominant share	resource shares	dominant share	resource shares	dominant share			
Tenant a	<2/24,4/36,3/54>	4/36	<0,0,0>	0	<0,0,0>	***0***	2/24	4/36	3/54
Tenant c	<2/24,4/36,3/54>	4/36	<0,0,0>	***0***	<1/24,3/36,6/54>	6/54	3/24	7/36	9/54
Tenant b	<2/24,4/36,3/54>	***4/36***	<3/24,2/36,6/54>	3/24	<1/24,3/36,6/54>	6/54	6/24	9/36	15/54
Tenant a	<4/24,8/36,6/54>	8/36	<3/24,2/36,6/54>	3/24	<1/24,3/36,6/54>	***6/54***	8/24	13/36	18/54
Tenant c	<4/24,8/36,6/54>	8/36	<3/24,2/36,6/54>	***3/24***	<2/24,6/36,12/54>	12/54	9/24	16/36	24/54
Tenant b	<4/24,8/36,6/54>	***8/36***	<6/24,4/36,12/54>	6/24	<2/24,6/36,12/54>	12/54	12/24	18/36	30/54
Tenant a	<6/24,12/36,9/54>	12/36	<6/24,4/36,12/54>	6/24	<2/24,6/36,12/54>	***12/54***	14/24	22/36	33/54
Tenant c	<6/24,12/36,9/54>	12/36	<6/24,4/36,12/54>	***6/24***	<3/24,9/36,18/54>	18/54	15/24	25/36	39/54
Tenant b	<6/24,12/36,9/54>	***12/36***	<9/24,6/36,18/54>	9/24	<3/24,9/36,18/54>	18/54	18/24	27/36	45/54
Tenant a	<8/24,16/36,12/54>	16/36	<9/24,6/36,18/54>	9/24	<3/24,9/36,18/54>	***18/54***	20/24	31/36	48/54
Tenant c	<8/24,16/36,12/54>	16/36	<9/24,6/36,18/54>	9/24	<4/24,12/36,24/54>	24/54	21/24	34/36	54/54

TABLE 1 Resource allocation example using our approach: Each row represents our DRF to make a scheduling decision. Each row shows the share of each tenant for each kind of resource, the tenant's dominant share, and the fraction of each resource allocated so far. Our DRF repeatedly selects the tenant with the lowest dominant share (indicated in bold and red) to launch a task, until no more tasks can be allocated. In the beginning, Tenant *a* and Tenant *c*'s dominant share fraction are smallest and the same (1/9), and Tenant *a* is chosen first. Next iteration, Tenant *c*'s dominant share fraction is the smallest, so Tenant *c* is selected. In the third iteration, Tenant *b*'s dominant share fraction is 0 so far and only Tenant *b* has not been given resources, so *b* is selected. We repeat the iteration until the cluster is saturated. Even if at the first iteration, when Tenant *c* is being selected, we could also get the same results.

<1/12,1/9,1/18> <1/8,1/18,1/9> <1/24,1/12,1/9>
<2/24,4/36,3/54> <2/24,4/36,3/54> <2/24,4/36,3/54>
task, until no more tasks can be allocated.
Table: Using our resource allocation policy in a cluster of 24 CPUs, 36 GB RAM, 54 disks with 3 users running tasks that require <2 CPU, 4 GB, 3 disk>, <3 CPU, 2 GB, 6 disk>, and <1 CPU, 3 GB, 6 disk> respectively under the resource sharing degree is 1. Each row corresponds to our DRF making a scheduling decision. A row shows the shares of each user for each kind of resource, the user's dominant share, and the fraction of each resource allocated so far. Our DRF repeatedly selects the user with the lowest dominant share (indicated in bold and red) to launch a task, until no more tasks can be allocated. (P.S. allo.: allocation dom.:dominant res.:resources).

In this example, we consider a special scenario that User A and User C's dominant share fraction are the same. Compared to those scenarios that all the users' dominant share fraction are different, this case will indicate the generality of our algorithm more accurately, since the realization of this policy depends on the bottom component. Even if the dominant share fraction of different users is same, it will not influence our algorithm. In the beginning, User A and User C's dominant share fraction are smallest and same (1/9), and User A is chose first. Next iteration, User C's dominant share fraction is

5.3 | Overview of Our Enhanced Fairer DRF Allocation Algorithm

Our enhanced fairer DRF allocation algorithm is added to the YARN Capacity Scheduler to consider CPU, memory and disk I/O resources, as shown in Figure 2. The queue from YARN's scheduling unit is a logical collection of applications submitted by

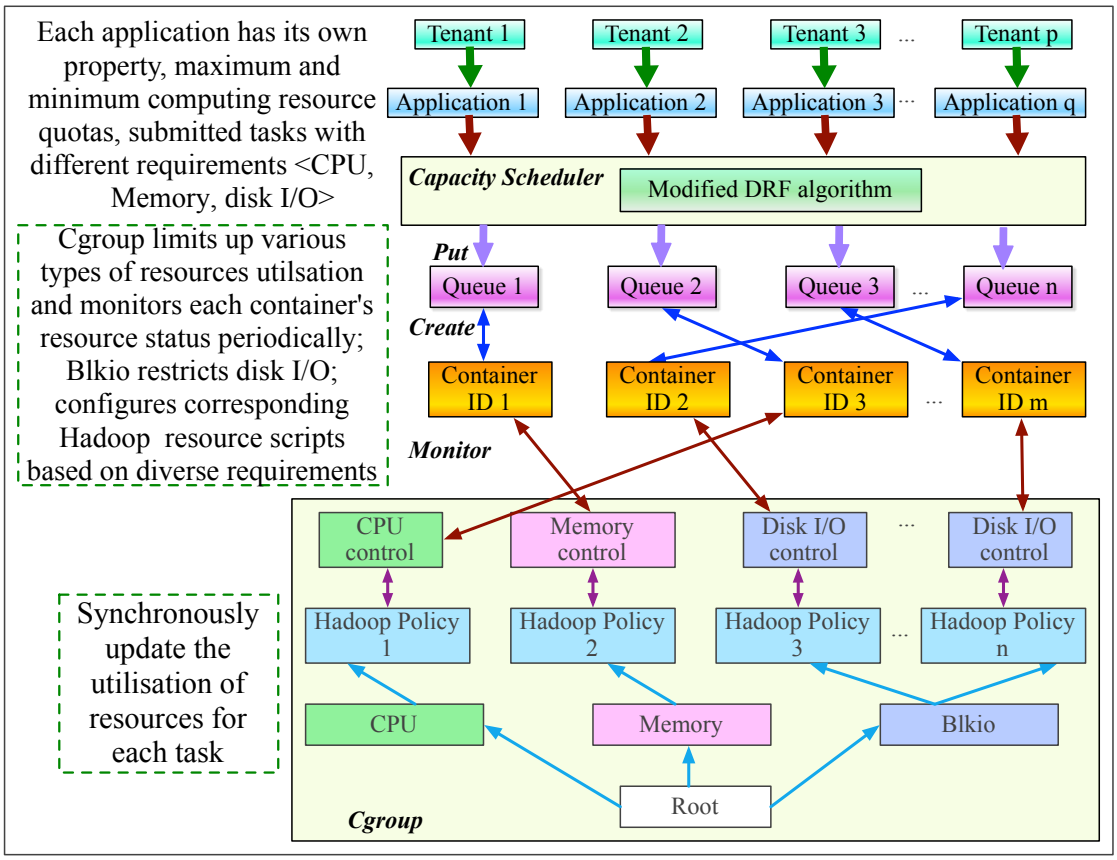

FIGURE 2 Our enhanced fairer DRF allocation algorithm

diverse tenants and can also be regarded as a logical view of the resources on physical nodes. The capacity of each queue specifies the percentage of cluster resources that is available for applications submitted to the queue. Tenants use YARN to orchestrate applications with differing resource requirements and to arbitrate all kinds of resources. Capacity Scheduler is enabled to allocate resources using the Dominant Resource Calculator based on our improved DRF model, where our algorithm is invoked.

Capacity scheduling represents one aspect of YARN resource management capabilities that includes *Cgroup*, node labels, archival storage, and memory as storage. In our allocation algorithm, *Cgroup* is used with capacity scheduling to constrain and manage CPU processes and 'blkio' configures different resource provision for jobs based on diverse tenants' requests. A container is a logical bundle of resources bound to a particular cluster node[24]. Containers with different resource configurations grant rights to corresponding tasks and provide specific amount of resources to process them. *Cgroup* also monitors the running status and allocates resources for each tenant and dynamically controls and tunes I/O and other resource allocations to isolate data. With *Cgroup* strict enforcement turned on, each task gets the only resources it asks for. Without *Cgroup* turned on, the DRF scheduler will do its best to balance allocations out, but unpredictable behaviour may occur. Our algorithm can force some allocation to specified disks via I/O matching. For example, high I/O tasks are assigned to a disk partition with a high I/O capability[24].

5.4 | Pseudo Code of Our Enhanced Fairer DRF Allocation Algorithm

Algorithm 1 shows the pseudo code of our modified DRF algorithm. A task is submitted with different resource demands, which is depicted by a 3-dimensional demand vector. In the naïve DRF algorithm, only CPU and memory can be regarded as a dominant share. However, our algorithm adds disk I/O to this resource demand vector. Dos_m is used to denote the demand vector of the next task that tenant m wants to launch (*Line:3*). The notations used in this work are listed in the Appendix. At each iteration, the scheduler selects the task with the lowest dominant share ready to run. If that tenant's next ready task requirement (Dem_m) can

Algorithm 1 Our Enhanced Fairer DRF Algorithm

1: $Res = (r_1, r_2, ..., r_p) \rightarrow$ total resources capacities
2: $Com = (c_1, c_2, ..., c_p) \rightarrow$ consumed resources, initial value = 0
3: Dos_m $(m = 1, 2, ...q) \rightarrow$ tenant m's dominant shares, initial value = 0
4: $All_m = (a_{m,1}, a_{m,2}, ..., a_{m,p})(m = 1, 2, ..., q) \rightarrow$ the resources allocated to tenant m, initial value = 0
5: $Lev = i$ $(i = 1, 2, 3) \rightarrow$ receive amount of resources based on the level. The lower level is, the more dominant resource is.
6: **Select** tenant m with the lowest dominant share Dos_m
7: $Dem_m \rightarrow$ the demand of the next task that tenant m wants to launch
8: **if** $Com + Dem_m \leq Res$ **then**
9: $Com = Com + Dem_m \rightarrow$ update consumed resources
10: $All_m = All_m + Dem_m \rightarrow$ update m's resource allocation
11: $Dos[] = sort^{p}_{n=1}(a_{m,n}/r_n)$
12: $Dos_m = Dos[Dos.length - Lev] \rightarrow$ determine dominant share degree
13: **else**
return $\rightarrow$ the cloud cluster is full
14: **end if**

be satisfied, then the task will be executed. Next, the scheduler updates the tenant's resource utilisation and adds the requirement of last running task (Dem_m) to tenant m's total allocated resources (All_m). When some tasks are finished, the tenants release their corresponding resources and recalculate the tenants' total allocated resources. Our algorithm uses $Lev = i$ $(i = 1, 2, 3)$ to determine the degree of resource sharing: high, medium, and low (*Line:5*). The tenant's task with lowest dominant share Dem_m is ready to run (*Lines:6-7*). If the inequality in *line:8* is satisfied, the task is executed and then resource usage is updated for All_m (*Lines:9-10*). When tasks finish, related allocated resources are released. Our algorithm can determine different degrees of dominant share. If $Lev = 3$, the smallest amount of needed resource for a tenant is set as its dominant share, which realises the lowest dominant resource share (*Lines:11-12*). If $Lev = 2$, we choose the middle amount of needed resource as dominant resource share for a tenant. If $Lev = 1$, the most amount of required resource is chosen as dominant resource share. Our method uses a binary heap to store each tenant's dominant share and then uses an array sort to store and determine the degree of the dominant share. Each scheduling decision takes $O(n \log n)$ time for n tenants.

6 | A DATA LOCALITY AWARE TASK SCHEDULER

Scheduling optimisation is an important approach to improve data locality by attempting to locate a task and its related data on the same node. We describe a novel optimised data locality aware task scheduler for balancing time consumption and network bandwidth traffic – DLAforBT – to improve data locality for tasks, with the optimal placement policy exhibiting a threshold-based structure, improving resource utilisation and throughput and shortening completion time[64]. DLAforBT[64] transforms the data locality scheduling problem into the well-known maximum weighted bipartite matching (MWBM) graph problem; uses a judgment mechanism to dynamically adjust task allocation and an embedded precise prediction model to determine moving computation or moving data, which demonstrates the benefits it offers to cloud systems; and ranks a list of idle computation capacity nodes based on descending order and gives different priorities to nodes, to maximise resource usage [‡]. This scheduler focuses on fine-grained task scheduling for high-latency applications. DLAforBT also makes approximations when scheduling and trading off many of the complex features supported by sophisticated schedulers in order to provide higher scheduling throughput. Our DLAforBT mainly pays attention to data or computation intensive jobs. When new jobs come, DLAforBT can allocate them to appropriate nodes. This scheduler can process both batch jobs and interactive jobs.

[‡]More details of this scheduler is described in our previous work[64]

6.1 | Data Placement Modelling

Our task scheduling is a bipartite graph matching which maps the knowledge of data block distribution and relevant performance of computing nodes. In this work, the cloud storage system is based on Hadoop distributed file system (HDFS)[7]. HDFS is based on the Google file system, which is designed with the master-slave architecture and can run on a cluster of computers that spread across many racks[7]. Each input file in HDFS is split into several fixed-size data blocks. Data blocks are distributed across nodes, and each of them is generally replicated as 3 times for data coverage and fault tolerance, in which placing one replica of the data block on a node in the same rack and the other replica on a node in another rack. We consider to schedule a set of tasks in a cloud system. As shown in Figure 3, there are p ($p = 6$) tasks and q ($q = 3$) nodes (servers), where each task should be processed on one node. A job will not be completed until all the sub tasks are finished. We rank nodes based on idleness utilisation of computation capacity in a descending order. Usually, we choose the node on the top of the ranking list with a required replica. The node with the highest available resource utilisation will be the first option to process tasks. This problem has been stated as NP-complete in strict cases[54].

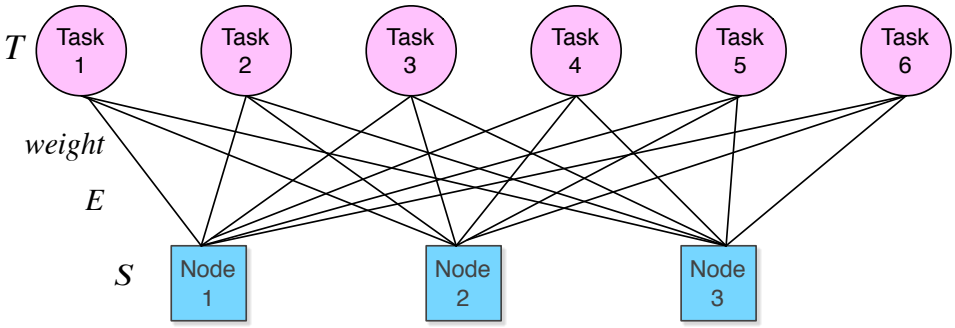

FIGURE 3 A data placement model $G(T \cup S, E)$

The data placement is modeled by a weighted bipartite graph $G = (T \cup S, E)$, let $p = |T|$ and $q = |S|$, where, T is the set of tasks, S is the set of cluster nodes, and $E \subseteq T \times S$ is the set of edges between T and S [§]. wei_i is the set of edges' weights and indicates remaining available resource utilisation of nodes. Resource utilisation of Node i is u^i_{res}. Thus the available resource utilisation of Node i is $u^i_{ava\_res} = 1 - u^i_{res}$, which is denoted as weight $wei(*, i)$ of connected node i. Edge $e(t, s)$ denotes that the input data blocks of task $t \in T$ is placed on the node $s \in S$. $S^G_{pre}(t)$ is the set of task t's preferred nodes in G. Assuming that $S^G_{pre}(t) \geqslant 1$, it means all of the nodes in T have at least 1 degree. When the degree is larger than 1, task t selects node s which has maximum weight wei_s, $s \in S^G_{pre}(t)$, under the same circumstance that several nodes all have the same required data blocks.

The task allocation can be transformed to the maximum weighted bipartite matching problem. Resource allocation is simply described as a function $f : T \to S$ that allocates task t to node $f(t)$. Defining α is the allocation for task t. Under current total allocation $\sum_{t \in T} \alpha(t)$ in the cloud, some nodes are assigned for some tasks. Assuming the tasks are performed sequentially on one node[54]. To avoid overload of some nodes and hotspots, and balance loads among the nodes, there exist multiple edges $e(t, s)$, between task t and nodes. Choose the node which has the maximum weight $wei(t, s)$ and then mark s as $\alpha(t)$. Task t is allocated to Node $\alpha(t)$. Node $\alpha(t)$'s resource utilisation is $u^{\alpha(t)}_{res}$, and the available resource utilisation of Node $\alpha(t)$ is $u^{\alpha(t)}_{ava\_res} = 1 - u^{\alpha(t)}_{res}$. Under α, task t is **local** iff $\alpha(t)$ is defined and there is an edge $e(t, \alpha(t))$ in data placement graph G. Otherwise, task t is **remote**. Let loc be the number of local tasks and rem be the number of remote tasks, respectively.

6.2 | Transfer Time

Each job j is associated with a data retrieval limit, which is represented as the maximum number of hops h_j allowed to access a data set. The data retrieval cost restricts the distance between the processing node with task t and the storage node with the data set of the task[36]. When node s runs task t, the transfer capability of node s is defined as follows[36]. In[36], it dedicates the data

[§]The notations are listed in the Appendix.

to the task is over TCP protocol. In TCP, the transfer throughput is dependent on TCP window size, the size of transferred data set, and the round trip delay time (RTT). Task t's transfer capacity cap_t can be estimated as follows:[36]

$$cap_t = \frac{tw_t}{rtt_t(n_i, n_j)} \tag{2}$$

where tw_t is TCP window size of an initiated TCP connection in task t which specifies the maximum number of data bytes to be received, $rtt_t(n_i, n_j)$ is the round-trip delay time for TCP connection between node n_i and node n_j. With Equation (2), if task t receives data block d_t with size $|d_t|$, data transfer time $T_t^{tran}(d_t)$ can be estimated as[36]:

$$T_t^{tran}(d_t) = \frac{|d_t|}{cap_t} = |d_t| \times (\frac{rtt_t(n_i, n_j)}{tw_t}) \tag{3}$$

6.3 | Remaining Execution Time Estimation

We design a model to estimate the remaining time of running jobs. To judge moving computation or data, we must estimate the resource releasing time (or the remaining execution time of tasks). Machine learning (ML) has been explored for performance prediction in individual compute nodes and distributed systems. The work in[68] presents a grey-box approach based on ML regression techniques for performance prediction in cloud environment. The use of ML techniques opens up the possibility to include a large number of node-specific factors that affect performance, thereby facilitating the capture of resource heterogeneity and contention[68]. The results in[68] have shown that supervised learning regression techniques, such as Multilayer Perceptron (MLP), have good prediction accuracy and prediction computation time for the conditions studied. MLP model as a back-propagation method based on neural networks is precise enough to estimate the remaining execution time of tasks. We chose MLP in our implementation, which proved to be satisfactory in our experiments. In this work, Keras[69] MLP API is invoked to realise execution time prediction. We use ReLu (Rectified Linear Units) as our activation function.The ReLu activation function $f(x)$ is improved as:

$$f(x) = \begin{cases} \frac{x+|x|}{2}, \alpha = 0 \\ \frac{1+\alpha}{2}x + \frac{1-\alpha}{2}|x|, \alpha \neq 0 \end{cases} \tag{4}$$

where x is symbolic tensor to compute the activation function, and α is scalar of tensor, which is optional, aiming to slope for negative input, usually between 0 and 1. The default value of 0 will lead to the standard rectifier, 1 will lead to a linear activation function, and any value in between will give a leaky rectifier.

6.4 | Cost of tasks

To capture locality, we define a cost function for a task that measures the sum of the execution time and the input data transfer time. $C(t, \alpha(t))$ denotes the cost of task t which is processed on Node $\alpha(t)$. It is defined as follows:

$$C(t, \alpha) = \begin{cases} C_{loc}, \text{ if } t \text{ is local in } \alpha \\ C_{rem}, \text{ otherwise} \end{cases} \tag{5}$$

C_{loc} is local cost and C_{rem} is remote cost. If all the required data blocks are in the same node, the time of reading input data from the local disk is negligible. Thus, C_{loc} indicates the execution time of task t, while C_{rem} is the sum of execution time and data transfer time. **Remote** tasks compete for network resources, so remote cost C_{rem} grows with the increment of the number of **remote** tasks. Assuming that

$$C_{rem}^{\alpha} = C_{rem}(rem), \qquad C_{loc}^{\alpha} = C_{loc}(loc) \tag{6}$$

where, $C_{rem}(\bullet)$ and $C_{loc}(\bullet)$ are monotone increasing functions, rem is the number of **remote** tasks and loc is the number of **local** tasks. Jobs are submitted and are divided into small tasks which are processed in parallel. Consequently, the total completion time of jobs T_{total}^{com} is: $\sum_{t \in T} C(t, \alpha)$. The task allocation problem can be transformed to the maximum weighted bipartite matching problem. This problem finds a complete allocation that minimises the total completion time, reduces the number of **remote** tasks rem, and improves system's throughput and locality rate. Edge $e(t, \alpha(t))$ indicates that Node $\alpha(t)$ has task t's data. Putting t on Node $\alpha(t)$ or another node depends on available resource utilisation of Node $\alpha(t)$: $u_{ava\_res}^{\alpha(t)}$, and the judgment mechanism which works by evaluating data transfer time $T_t^{tran}(d_t)$ and resource releasing time $W_{\alpha(t)}^{rel}$.

6.5 | Overview of DLAforBT

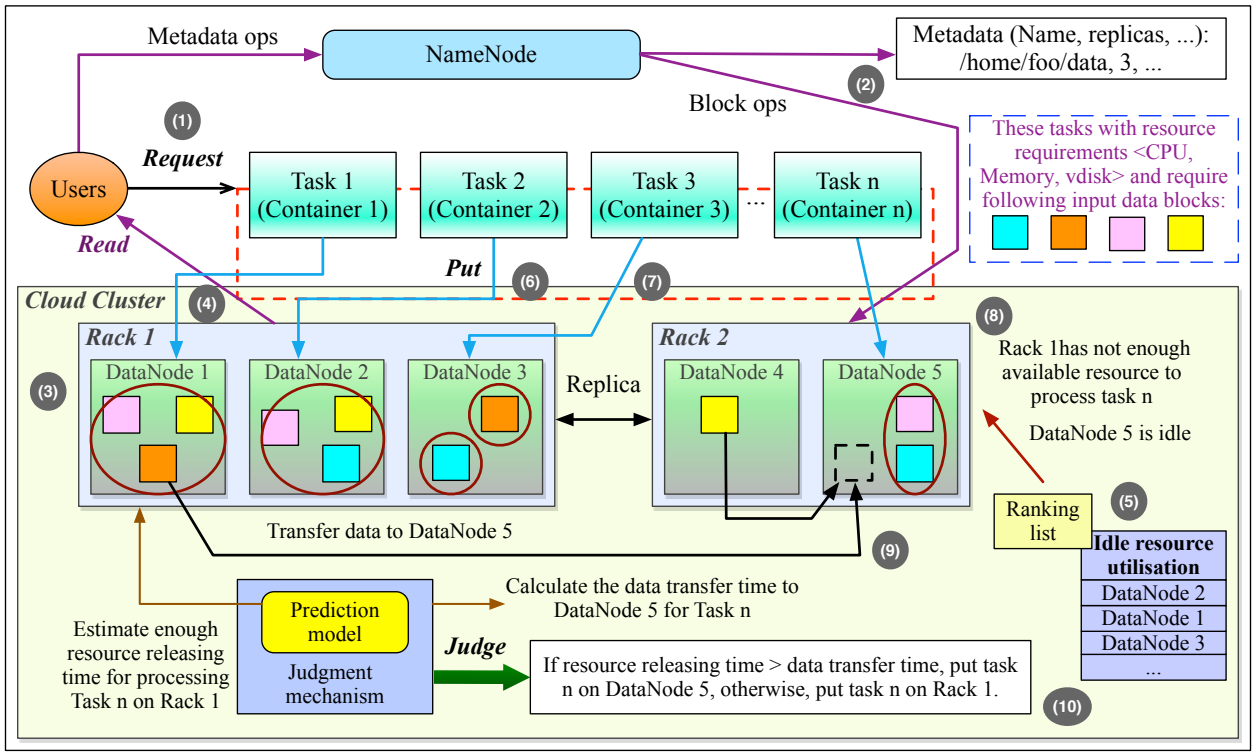

FIGURE 4 DLAforBT scheduler architecture: moving computation or moving data?

DLAforBT finds an efficient solution in time $O(n^3)$. On investigating a feasible solution, one crucial obstacle is to gain a weighted perfect matching when nodes' resource utilisation varies frequently. Figure 4 presents our DLAforBT's architecture, including a task allocation policy, a ranking list of idle resource usage for cluster nodes, a judgment mechanism and an inside prediction model. Users submit jobs to the cloud and the jobs are divided into n tasks (1). NameNode manages the file system metadata (2). DataNodes spread across multiple racks and store data (3). These n tasks need data blocks that are mostly stored in DataNode 1 and DataNode 2 on Rack 1, so most of these tasks should be run on Rack 1 and the corresponding containers should also be created and put on Rack 1. Task 1 is deployed on DataNode 1 which has 3 data blocks (4). It ranks DatasNodes based on a descending order of idle resource utilisation (5). Task 1 occupies some resources of DataNode 1. So DataNode 2's idle resource utilisation is higher than that of DataNode 1, and DataNode 2 has higher priority. Thus even if DataNodes 1 and 2 both have same data blocks, task 2 should be allocated to DataNode 2 (6). DataNodes 1 and 2 are busy in processing tasks and the nodes are nearly full, so the new coming task 3 cannot gain sufficient resource from these 2 DataNodes, even if they have the most relevant data blocks. At this stage, the system should consider waiting for resources to be released to allocate task 3 on DataNode 1 or 2, or putting task 3 to another node. DLAforBT estimates enough resource releasing time of DataNode 1 or 2 and calculates data transfer time to DataNode 3 which has the second most corresponding data of task 3. If our judgment mechanism finds resource releasing time is larger than data transfer time, then task 3 will be allocated to DataNode 3 (7). If all nodes on Rack 1 are busy, then task n cannot get enough resource from Rack 1. DataNode 5 on Rack 2 is idle, but it only contains 2 required data blocks (8). Thus, to process task n, the scheduler would need to transfer 1 data block (orange) from Rack 1 and 1 data block (yellow) from DataNode 4 to 5 (9). Our judgment mechanism finds that the data transfer time to DataNode 5 is smaller than the waiting time for releasing resource from DataNodes 1 and 2 (10). Thus task n is put on DataNode 5.

Algorithm 2 DLAforBT Scheduler

Input: A set J of jobs with different resource requirements and a set S of nodes with heterogeneous performance
Output: Data locality aware scheduling for the tasks of J

1: **while** (jobs run their sub tasks) **do**
2: $Res^n_{rem} \rightarrow$ remaining available resources of node n; $Dem_t \rightarrow$ the resource demand of $task_t$
3: **for** (each job j of J in the queue) **do**
4: Partition job j into sub tasks T
5: **end for**
6: **// Bipartite graph modelling**
7: Invoke Algorithm 3 to model the connections between T and S
8: $T \leftarrow$ collect the running tasks of such jobs
9: **for** (each task t of T) **do**
10: $f(t) \leftarrow$ find the feasible nodes of t
11: Collect $f(t)$ in S
12: **end for**
13: Form a weight bipartite graph based on T and S sets
14: **// Scheduling problem transformation**
15: Apply the maximum weighted bipartite matching (MWBM) to obtain the optional task scheduling of T
16: **// Judgment mechanism**
17: Invoke Algorithm 4 to determine moving data or computation
18: After MWBM, $t \rightarrow$ node $f(t)$
19: **if** ($Res^{f(t)}_{rem} < Dem_t$) **then**
20: Apply judgment mechanism to decide t's placement: put t on $f(t)$ or move t to another node
21: **else**
22: Put t on $f(t)$
23: **end if**
24: Create a container con_t for t, and con_t has Dem_t resources
25: **end while**

6.6 | Pseudo Code of DLAforBT

Algorithm 2 describes the approach of DLAforBT. When a job is submitted, the job is associated with data retrieval and placed in a queue. After partition, the job is divided into tasks (*Lines:1-4*). Whenever one or more jobs in the ready queue run their tasks simultaneously, these running tasks are collected in task set T. Next, DLAforBT finds the feasible nodes of each running task (*Lines:7-8*). If node s is a feasible node of task t, node s will provide the appropriate execution performance to meet task t's requirements and data retrieval of the original job. Work in [36] declared that job characteristics can be obtained by job profiling. This characteristic provides inference information between input data blocks and tasks. Using the inference information, we can precisely know that the locations of data blocks linked to tasks. For all running tasks, the feasible nodes found are kept in set S (*Lines:9-12*). Based on sets T and S, we construct a weighted bipartite graph to represent the scheduling (allocation) relationship between T and S (*Line:13*). Yet, even if finding appropriate node $f(t)$ for task t, $f(t)$ may not provide enough resources for t (*Lines:14-15*). We use our judgment mechanism to decide whether moving t to other nodes, or waiting on $f(t)$ until $f(t)$ has enough resources to run t (*Lines:17-23*). A corresponding container con_t with a timestamp is created for task t (*Line:24*).

6.6.1 | Weighted Bipartite Graph Formation

The data volume is increasing vastly. Today's data analytics clusters are running ever long and high-fanout jobs. These jobs arise not only due to frameworks targeting the latency, but also as a result of breaking long-running batch jobs into a large number of short tasks which tries to improve fairness and mitigate stragglers [70]. Multiple submitted jobs from multiple tenants are divided into tasks, which present a difficult scheduling challenge. To run tasks efficiently in parallel, these tasks must be allocated to appropriate nodes and scheduling decisions must be made at very high throughput. These tasks are going to be immensely distributed in volume. To improve the efficiency of task scheduling, we also allocate individual tasks to different

intensive queues according to their different dominant resource consumptions. Such task allocation is a significant issue of the scheduling problem[71].

To efficiently solve this task allocation problem, we use a weighted bipartite graph to model all feasible allocation cases between tasks and nodes. Given a weighted bipartite graph, the maximum weighted bipartite matching problem is to select a maximum number of adjacent weighted edges, so that the total weight of the selected edges is maximised. Kuhn-Munkres (KM) algorithm is a combinatorial optimisation algorithm that solves the assignment problem in polynomial time which anticipates latest primal-dual methods[72]. KM algorithm is widely used in weighted bipartite graph model to realise task scheduling, such as[54,55,73]. The KM algorithm[72] is an efficient way to find the maximum weight perfect matching in a weighted bipartite graph and find a good feasible labeling that remains enough edges in graph. If an edge $e(t, s)$ is selected, it represents that task t will be assigned to node s to run. Non-adjacent edge selection makes that the selected edges do not have required data blocks or enough resources to process the tasks. By minimising the number of non-adjacent edges, the tasks should be allocated to the nodes which have relevant data blocks as much as possible. By maximising the weights of adjacent edges, the nodes which have relevant data to process corresponding tasks having maximum resource utilisation should be the first option. The time complexity of KM algorithm in our work achieves $O(n^3)$ running time.

Algorithm 3 Maximum Weighted Bipartite Graph Formation

Input: A set T of tasks, a set S of heterogeneous nodes
Output: A weighted bipartite graph $BG = (T \cup S, E)$

```
1:  for (each task t of T) do
2:      for (data locality nodes of t) do
3:          Locality nodes are with the h-hop characteristic, which are determined by Equation (7)
4:          if (node s meets task t's data requirements) then
5:              S ← S ∪ s
6:              wei(t, s) ← The edge weight is associated with idleness utilisation for computation capacity of s
7:              E ← E ∪ (t, s), (t, s) is the edge between t and s
8:          end if
9:      end for
10: end for
11: for (each task t of T) do
12:     max_wei(t, s) ← Find the maximum edge weight from all the edges between s and t
13: end for
```

Algorithm 3 shows maximum weighted bipartite graph modelling. Assuming that nodes n_s and n_d with resources to run task t and with the input data blocks of t, respectively. The following equation restricts the data retrieval limit between nodes n_s and n_d based on a hop limit h[36].

$$Dist(n_s, n_d) \leqslant h \tag{7}$$

where $Dist(n_s, n_d)$ is the number of hops between nodes n_s and n_d (*Line:3*). The number of hops denotes the number of switches involved in the data transfer path between n_s and n_d. T and S are the two disjoint node sets of BG. If node s of S is a feasible node of task t of T, there is a corresponding edge $e(t, s)$ in BG (*Lines:4-5*). An edge is associated with an edge weight. We label the weight $wei(t, s)$ as the available resource utilisation of node s: $u^i_{ava\_res} = 1 - u^i_{res}$ (*Lines:6-7*). In BG, it is possible that some nodes act as the feasible nodes of task t. We only choose the node with the maximum weighted edge (*Lines:11-13*).

6.6.2 | Judgment Mechanism

The network is shared between nodes, which is also a well-known bottleneck. Sometimes, required data blocks are quite large. Transferring these data blocks to another node via network may occupy too much bandwidth and introduce network bottleneck. To reduce the influence of network bandwidth limitation, we build a judgment mechanism to decide whether moving data blocks from one node to another node or moving tasks (code) from one node to another node.

Algorithm 4 presents the pseudo code of our judgment mechanism. When there are not enough resources in the appropriate node $f(t)$ for processing task t, we calculate the required data transfer time for task t, $T^{tran}_t(d_t)$, and estimate the required amount

of resource releasing time of $f(t)$, $W^{rel}_{f(t)}$ (*Lines:1-4*). That means we need to predict the remaining execution time for running tasks, $W^{soj-rem}$. We use a multilayer perceptron prediction model to train historical data, and gain an estimated execution time of tasks, W^{soj}. We can capture previous running time of tasks, $W^{soj-alr}$, to get

$$W^{rel}_{f(t)} = W^{soj-rem} = W^{soj} - W^{soj-alr} \tag{8}$$

When resource releasing time $W^{rel}_{f(t)}$ is less than data transfer time $T^{tran}_t(d_t)$, task t costs less tradeoff on $f(t)$, so we put t on $f(t)$ (*Lines:5-7*). Otherwise, task t spends more time on waiting for resources on $f(t)$, so we move data to another node (*Lines:8-9*). Network bandwidth also has an impact on node selection. Sinbad has adapted network resource utilisation to navigate the data replica location selection[74]. When moving tasks to other nodes, considering the network bandwidth consumption, firstly, we select the node from the same rack according to the ranking list of idleness utilisation for computation capacity. If all these nodes on the same rack do not have enough resource to process task t, then we select the nodes on different racks based on the ranking list.

Algorithm 4 Judgment Mechanism

Input: A set S of nodes linked to a set T of tasks
Output: Putt t on their relevant nodes $f(t)$ or move t to other nodes

1: **while** $(Dem_t < Res^{f(t)}_{rem})$ **do**
2: Calculate t's data transfer time $T^{tran}_t(d_t)$
3: Call Keras API to train tasks → get the execution time of tasks running on $f(t)$
4: Calculate $f(t)$'s resource releasing time $W^{rel}_{f(t)}$ = remaining execution time of task on $f(t)$
5: **if** $(W^{rel}_{f(t)} < T^{tran}_t(d_t))$ **then**
6: Put t on $f(t)$ to wait for resources
7: **else**
8: find a neighbouring node $n_{nei} \leftarrow (Res^{n_{nei}}_{rem} > Dem_t)$
9: Move t to n_{nei}
10: **end if**
11: **end while**

7 | A MULTI-TENANT DEADLINE CONSTRAINED JOB SCHEDULER

For real-time applications, deadline is a major criterion in judging QoS. We present a real-time, preemptive, deadline constrained scheduler using queuing theory – PDSonQueue – which enables better QoS, such as reducing deadline-based SLA violation, shortening jobs' completion time and improving system's throughput and performance[65]. PDSonQueue, as a dynamic priority real-time greedy job scheduler builds an M/M/n mathematical queuing model to accurately predict jobs' execution time and waiting time, where jobs arrive by following a stochastic process and request resources. This scheduler introduces a novel concept – "Earliest Maximal Waiting Time First (EMWTF)" to fine tune job scheduling to guarantee the job being accomplished within the deadline. The deadline constrained jobs are scheduled preemptively from low priority jobs with the intent of maximising the number of jobs completed within the deadlines, while allowing system's resources to be shared by other regular jobs [¶].

7.1 | Earliest Maximal Waiting Time First (EMWTF)

In cloud computing, deadline is one of the most significant QoS features and sorting jobs based on deadlines is critical. Indeed, the long jobs may not be affected if they are delayed in order of seconds, while short jobs which can also be latency sensitive have tighter SLO bounds. However, no matter long jobs or short jobs, they both need to be finished before their deadlines, regardless of the delay effect. We use deadline to determine the jobs' importance.

[¶]More details of this scheduler is described in our previous work[64]

Although EDF algorithms are popular in guaranteeing job deadlines in real-time systems, they are not effective in dynamic cloud environments, especially a Hadoop cluster with dynamic resources[43]. Physical resources and workloads are heterogeneous in cloud cluster. Each application(job)'s execution time is also different. Even if for the same application, the jobs may need various data blocks, so the execution time is diverse. Some jobs may be long jobs and others may be short ones. A long job has a longer deadline than that of short job, but the long job's execution time is also longer than that of the short job. Thus, long jobs' maximal waiting time (the difference between deadline and execution) may be shorter than the waiting time of short jobs. That means the long job may have less time in the queue to wait to run. For instance, job ***a*** is PI estimation with 180 seconds (secs) deadline requirement, and job ***b*** is data migration application with 600 secs deadline requirement and its data size is 10 GB. Job ***a*** may need 100 secs to execute, and its maximal waiting time is about 80 secs, while job ***b*** may need 550 secs to complete this data migration, and its maximal waiting time is only about 50 secs. Compared with conversional deadline scheduling algorithm
such as EDF, job ***a*** with smaller deadline, so it should be ahead of job ***b*** in the deadline queue and be performed first. However, in our scheduler, we put job ***b*** first, since its waiting time is smaller than job ***a***. As long as we begin to run both of these 2 jobs during their maximal waiting time, both of them will be finished before their deadlines. Given job ***b*** has 50 secs buffer and job ***a*** has 80 secs buffer, it is necessary to schedule job ***b*** first.

Therefore, even if with a longer deadline, the long job still needs to be processed earlier than that of short job. Within maximal waiting time, a job gets the necessary resources to run, which will guarantee this job will be finished before its QoS specified deadline. Since execution time varies based on different job types and input data sizes, we introduce a new concept "Earliest Maximal Waiting Time First" (EMWTF) instead of EDF. If using EDF, short job should be put ahead of long job to be completed. This may result in that long job fails to get resources during the waiting time and cannot be executed on time. If using EMWTF, long job which has less waiting time should be put ahead to be processed before short job, while short job has more waiting time to gain resources, so it also can be finished before its deadline. Therefore, by sorting jobs based on EMWTF, more jobs would be completed before their deadlines.

As a cloud centre can obtain many server nodes, we use an M/M/n mathematic queuing model to model job scheduling to estimate service time, waiting time, assuming an exponential density function for the inter-arrival and service times. Assuming that we have had some historical data, such as jobs' execution log, including finished jobs' service time and waiting time, as in practice, many jobs were run in the past. So we can predict a job's maximal waiting time before a deadline job is executed. Within the maximal waiting time, a job will obtain the necessary resources, which will guarantee this job finishes before its QoS specified deadline.

7.2 | Mathematical Analysis of Job Scheduling

A cloud system contains n heterogeneous computing nodes, donated as $Node_1$, $Node_2$,..., $Node_n$. The submitted jobs are served by n nodes operating independently[75]. We assume the arrival of the jobs conforms to a Poisson process λ_i. The service rate is also assumed to be independent and exponentially with parameter μ_k[75], where [#]:

$$\mu_k = \min(k\mu, n\mu) = \begin{cases} k\mu, \ for \ 0 \le k \le n, \\ n\mu, \ for \ k > n \end{cases}$$

The mean service rate of computing node j is μ, and thus mean service rate of entire cloud system is $n\mu$. When $\frac{\lambda}{n\mu} < 1$, the theory[75] has proven that the cloud system being stable, marking $\rho_1 = \frac{\lambda}{\mu}$, $\rho = \frac{\lambda}{n\mu}$. The service is the same as the job arrival rate that follows the Poisson process. Fig. 5 shows the M/M/n queuing model in cloud computing.

The i^{th} job job_i is defined as $job_i = \{Dem_i, t_i^{arr}, t_i^{dea}\}$ or $job_i = \{Dem_i, t_i^{arr}, p_i\}$, where Dem_i indicates required resource, t_i^{arr} presents arrival time, t_i^{dea} indicates deadline of the job, and specially, p_i indicates the job is regular and there is no deadline constraint. A regular job has priority: low and high. p_0 means low priority and p_1 means high priority.

7.2.1 | Steady State Equation

The state set of the cloud system is $\Phi = \{0, 1, 2, ...\}$, so these balance equations can be derived by the state transition flow diagram of M/M/n queuing model depicted in Figure 6. It shows the probability of different system's status and servers' status. When the state is k $(0 < k \le n)$, k servers are busy and the remaining $n - k$ servers are idle. When the state is $k > n$, all the n servers are busy, and $k-n$ jobs are waiting for the service. Assume there are 2 waiting queues: deadline queue and regular queue.

[#]The notations are listed in the Appendix.

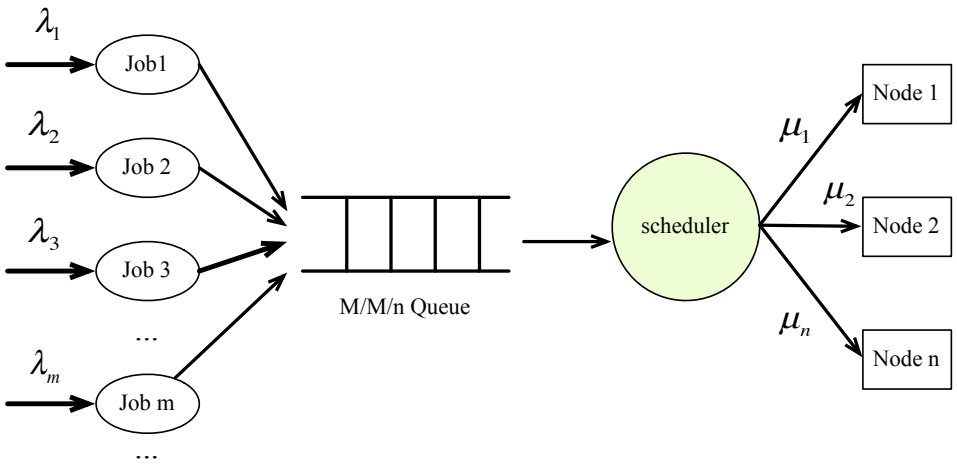

FIGURE 5 M/M/n queuing model for job performing in cloud computing

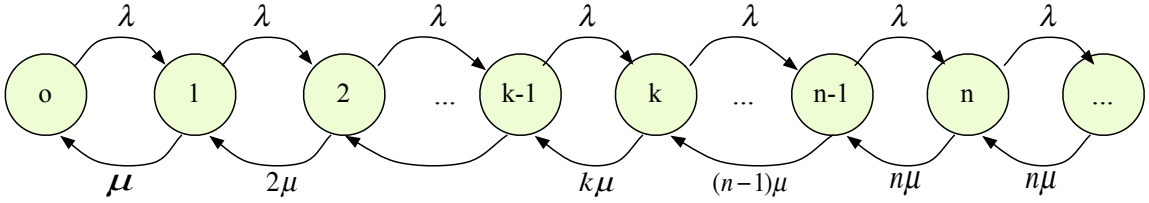

FIGURE 6 State transition flow diagram of M/M/n queuing model

When the system is stable, let $\rho = \frac{\lambda}{n\mu}$ and the stability condition $\rho < 1$. The stationary probability p_k for state k can be determined by solving the set of balance equations, which state that the flux into a state should be equal to the flux out of this state when the system is stationary[11]:

$$\begin{aligned}
&\text{When state } k = 0, \lambda p_0 = \mu p_1, && p_1 = \rho_1 p_0 = n\rho p_0;\\
&\text{When state } k = 1, \lambda p_1 = 2\mu p_2, && p_2 = \frac{\rho_1^2}{2!} p_0 = \frac{n^2}{2!}\rho^2 p_0;\\
&\text{When state } k = 2, \lambda p_2 = 3\mu p_3, && p_3 = \frac{\rho_1^3}{3!} p_0 = \frac{n^3}{3!}\rho^3 p_0;\\
&\qquad\qquad ... &&\\
&\text{When state } k = n-1, \lambda p_{n-1} = n\mu p_n, && p_n = \frac{\rho_1^n}{n!} p_0 = \frac{n^n}{n!}\rho^n p_0;\\
&\text{When state } k = n, \lambda p_n = n\mu p_{n+1}, && p_{n+1} = \frac{\rho_1^{n+1}}{n!n} p_0 = \frac{n^n}{n!}\rho^{n+1} p_0;\\
&\qquad\qquad ... &&\\
&\text{When state } k = n+r-1, \lambda p_{n+r-1} = n\mu p_{n+r}, && p_{n+r} = \frac{\rho_1^{n+r}}{n!n^r} p_0 = \frac{n^n}{n!}\rho^{n+r} p_0.
\end{aligned} \tag{9}$$

In general,

$$p_k = \begin{cases} \frac{\rho_1^k}{k!} p_0 = \frac{n^k}{k!}\rho^k p_0, & 0 \le k < n \\ \frac{\rho_1^k}{n!n^{k-n}} p_0 = \frac{n^n}{n!}\rho^k p_0; & k \ge n \end{cases} \tag{10}$$

According to regularity condition $\sum_{k=0}^{\infty} p_k = 1$, when $\rho < 1$, we can get $p_0 = (\sum_{k=0}^{n-1} \frac{\rho_1^k}{k!} + \frac{\rho_1^n}{n!}\frac{1}{1-\rho})^{-1}$.

7.2.2 | Mean Queue Length, Sojourn Time and Waiting Time

For estimation of mean queue length, sojourn time and waiting time, we declare some parameters and some notations (shown in TABLE 2).

TABLE 2 Summary of Notations

Notations	Description
L_{sys}	total number of jobs in cloud, including the ones waiting in the queue and being serviced. $\bar{L}_{sys}$ is the mean of L_{sys}
L_{wai}	total number of jobs waiting in the queue, excluding the ones being serviced. $\bar{L}_{wai}$ is the mean of L_{wai}
L_{ser}	total number of jobs being serviced. $\bar{L}_{ser}$ is the mean of L_{ser}
L	average number of jobs in the queue at any time
W_i^{wai}	the time that job_i waits in the queue, excluding the time that job_i spends on service. $\bar{W}_{wai}$ is the mean waiting time
W_i^{soj}	job_i's sojourn time, including the time that job_i waits in the queue and are in service. $\bar{W}_{soj}$ is the mean sojourn time
T_{ser}	average service time of a job, denoted as $\frac{1}{\mu}$

The mean number of jobs in the system is given as follows:

$$\bar{L}_{sys} = \bar{L}_{wai} + \bar{L}_{ser} = \bar{L}_{wai} + \rho_1 = \frac{\rho\rho_1^n p_0}{n!(1-\rho)^2} + \rho_1 \tag{11}$$

The mean number of jobs waiting in the queue is below:

$$\bar{L}_{wai} = \sum_{k=n}^{\infty}(k-n)p_k = \sum_{h=0}^{\infty} h p_{h+n} = \frac{\rho(n\rho)^n}{n!}p_0 \sum_{h=1}^{\infty} h\rho^{h-1} = \frac{\rho\rho_1^n}{n!(1-\rho)^2}p_0 = \frac{\rho_1^{n+1}}{(n-1)!(n-\rho_1)^2}p_0 \tag{12}$$

The following equation shows the mean number of jobs that are being serviced in the system:

$$\bar{L}_{ser} = \bar{k} = \sum_{k=0}^{n} k p_k + n\sum_{k=n+1}^{\infty} p_k = \sum_{k=0}^{n} k \bullet \frac{n^k}{k!}\rho^k p_0 + \sum_{k=n+1}^{\infty} n \bullet \frac{n^n}{n!}\rho^k p_0 = n\rho(\sum_{k=0}^{n-1} p_k + \sum_{k=n}^{\infty} p_k) = n\rho = \rho_1 \tag{13}$$

The variance of mean number of jobs waiting in the queue shows as follows. Since,

$$\begin{aligned} E(\bar{L}_{wai}^2) &= \sum_{k=n}^{\infty}(k-n)^2 p_k = \sum_{h=1}^{\infty} h^2 p_{h+n} = \sum_{h=1}^{\infty}\frac{h^2}{n!n^h}(n\rho)^{h+n}p_0 = \frac{(n\rho)^n\rho^2 p_0}{n!}\sum_{h=2}^{\infty} h(h-1)\rho^{h-2} + \frac{(n\rho)^n\rho p_0}{n!}\sum_{h=1}^{\infty} h\rho^{h-1} \\ &= \frac{2\rho^2\rho_1^n p_0}{n!(1-\rho)^3} + \bar{L}_{wai} = \frac{1+\rho}{1-\rho}\bar{L}_{wai} \end{aligned} \tag{14}$$

Thus,

$$\sigma^2(\bar{L}_{wai}) = E(\bar{L}_{wai}^2) - [E(\bar{L}_{wai})]^2 = \bar{L}_{wai}(\frac{1+\rho}{1-\rho} - \bar{L}_{wai}) \tag{15}$$

In addition,

$$E(L_{sys}) = E(L_{wai}) + E(L_{ser}) \quad , \quad E(W_{soj}) = E(W_{wai}) + E(T_{ser}) \tag{16}$$

we can easily get

$$E(T_{ser}) = \frac{1}{\mu} \tag{17}$$

If we only consider servers of the system, without regard of the waiting queues outside the servers, it is easy to observe that there are no losses. Therefore the arrival rate in this cloud system is λ, and the mean waiting time of each customer is $E(T_{ser}) = \frac{1}{\mu}$ [75]. When a queuing system reaches statistical equilibrium and $L = \lambda\bar{W}_{soj}$, $\bar{L}_{wai} = \lambda\bar{W}_{wai}$, our queuing system obeys Little's Law [75]. We can get the following equation:

$$\bar{W}_{wai} = \frac{\bar{L}_{wai}}{\lambda} = \frac{\rho_1^n p_0}{\mu n \bullet n!(1-\rho)^2} \quad , \quad \bar{W}_{soj} = \frac{\bar{L}_{sys}}{\lambda} = \bar{W}_{wai} + T_{ser} = \bar{W}_{wai} + \frac{1}{\mu} \tag{18}$$

The probability of that a job must wait in the queue to gain service from the cloud $P(waiting)$ is shown as follows:

$$P(Waiting) = \sum_{k=n}^{\infty} p_k = \sum_{k=n}^{\infty}\frac{n^n}{n!}\rho^k p_0 \tag{19}$$

whereas, $P(Waiting) = C(n,\rho)$, which is also referred as Erlang's C formula [75]:

$$C(n,\rho_1) = \sum_{k=n}^{\infty} p_n \bullet \rho^{k-n} = \frac{p_n}{1-\rho} = \frac{np_n}{n-\rho_1} \tag{20}$$

To calculate $E(L_{ser})$ for the M/M/n queue, through Little's Law, the mean number of busy servers is given below:

$$E(L_{ser}) = \frac{\lambda}{\mu} = \rho \tag{21}$$

To calculate $E(L_{wai})$, we assume two mutually exclusive and exhaustive events: $\{q \geq n\}$ and $\{q < n\}$, and then we get

$$E(L_{wai}) = E(L_{wai}|q \geq n)P(q \geq n) + E(L_{wai}|q < n)P(q < n) \tag{22}$$

To obtain $E(L_{wai}|q \geq n)$, noted that the evolution of the M/M/n queue during the time when $q \geq n$ is equal to that of M/M/1 queue with the arrival rate λ and the service rate $n\mu$. Therefore, the mean queue length of this kind of M/M/1 queue is equivalent to $\frac{1}{1-\rho}$, where $\rho = \frac{\lambda}{n\mu}$. Therefore,

$$E(L_{wai}|q \geq n) = \frac{\rho/n}{1-\rho/n} = \frac{\rho}{n-\rho} \tag{23}$$

Due to $E(L_{wai}|q < n) = 0$, $P(q \geq n) = C(n, \rho)$, we get

$$E(L_{wai}) = C(n, \rho)\frac{\rho}{n-\rho} \tag{24}$$

The following formulas calculate the distribution of waiting time and sojourn time. An arriving job has to wait if the number of arrival jobs in the system is at least n. The time when a customer is serviced is exponentially distributed with parameter $n\mu$. Consequently, if there are $n+j$ jobs in the system, the waiting time is Erlang distributed with parameters $(j+1, n\mu)$. By applying the theorem of total probability to the density function of waiting time, we get[75]

$$f_w(x) = \sum_{j=0}^{\infty} p_{n+j}(n\mu)^{j+1}\frac{x^j}{j!}e^{-n\mu x} \tag{25}$$

Substitute the distribution for the density function of the waiting time, we get

$$\begin{aligned} f_w(x) &= \frac{p_0(\frac{\lambda}{\mu})^n}{n!}n\mu e^{-n\mu x}\sum_{j=0}^{\infty}\frac{(\rho n\mu x)^j}{j!} = \frac{(\frac{\lambda}{\mu})^n}{n!}p_0 n\mu e^{-(n\mu-\lambda)x} = \frac{(\frac{\lambda}{\mu})^n}{n!}p_0 n\mu e^{-n\mu(1-\rho)x} = \frac{(\frac{\lambda}{\mu})^n}{n!}p_0\frac{1}{1-\rho}n\mu(1-\rho)e^{-n\mu(1-\rho)x} \\ &= P(Waiting)n\mu(1-\rho)e^{-n\mu(1-\rho)x} \end{aligned} \tag{26}$$

Thus, for the complement of the the distribution function, we have

$$P(W > x) = \int_x^{\infty} f_w(u)du = P(Waiting)e^{-n\mu(1-\rho)x} = C(n, \rho) \bullet e^{-\mu(n-\frac{\rho}{n})x} \tag{27}$$

The distribution function of waiting time can be written as:

$$F_w(x) = 1 - P(Waiting) + P(Waiting)(1 - e^{-n\mu(1-\rho)x}) = 1 - P(Waiting)e^{-n\mu(1-\rho)x} = 1 - C(n, \rho) \bullet e^{-\mu(n-\frac{\rho}{n})x} \tag{28}$$

If the number of arriving jobs in the system is smaller than n, then the jobs will immediately get serviced. Otherwise, the jobs have to wait and their sojourn times include waiting time and service time. By applying the law of total probability to the density function of sojourn time, $f_s(x)$ is given as follows:

$$f_s(x) = P(No\ waiting)\mu e^{-\mu x} + f_{w+ser}(x) \tag{29}$$

Whereas, the density function of sojourn time for the job that needs to wait first, $f_{w+ser}(x)$:

$$\begin{aligned} f_{w+ser}(z) &= \int_0^z f_w(x)\mu e^{-\mu(z-x)}dx = P(Waiting)n\mu(1-\rho)\mu\int_0^z e^{-n\mu(1-\rho)x}e^{-\mu(z-x)}dx \\ &= \frac{(n\rho)^n}{n!}p_0\frac{1}{(1-\rho)}n\mu(1-\rho)\mu e^{-z\mu}\int_0^z e^{-\mu(n-1-\frac{\lambda}{\mu})x}dx = \frac{(n\rho)^n}{n!}p_0 n\mu\frac{1}{(n-1-\frac{\lambda}{\mu})}e^{-z\mu}(1 - e^{-\mu(n-1-\frac{\lambda}{\mu})z}) \end{aligned} \tag{30}$$

Therefore,

$$\begin{aligned} f_s(x) &= (1 - (\tfrac{\lambda}{\mu})^n\frac{p_0}{n!(1-\rho)})\mu e^{-\mu x} + \frac{(\frac{\lambda}{\mu})^n}{n!}n\mu p_0\frac{1}{(n-1-\frac{\lambda}{\mu})}e^{-\mu x}(1 - e^{-\mu(n-1-\frac{\lambda}{\mu})x}) \\ &= \mu e^{-\mu x}(1 - \frac{(\frac{\lambda}{\mu})^n p_0}{n!(1-\rho)} + \frac{(\frac{\lambda}{\mu})^n}{n!}np_0\frac{1}{(n-1-\frac{\lambda}{\mu})}(1 - e^{-\mu(n-1-\frac{\lambda}{\mu})x})) = \mu e^{-\mu x}(1 + \frac{(\frac{\lambda}{\mu})^n p_0}{n!(1-\rho)}\frac{1-(n-\frac{\lambda}{\mu})e^{-\mu(n-1-\frac{\lambda}{\mu})x}}{(n-1-\frac{\lambda}{\mu})}) \end{aligned} \tag{31}$$

For the complement of the distribution function of the response time, we get

$$\begin{aligned} P(S > x) &= \int_x^{\infty} f_s(y)dy = \int_x^{\infty}\mu e^{-\mu y} + \frac{(\frac{\lambda}{\mu})^n p_0}{n!(1-\rho)}\frac{1}{(n-1-\frac{\lambda}{\mu})}(\mu e^{-\mu y} - \mu(n-\frac{\lambda}{\mu})e^{-\mu(n-\frac{\lambda}{\mu})y})dy \\ &= e^{-\mu x} + (\tfrac{\lambda}{\mu})^n p_0\frac{1}{n!(1-\rho)(n-1-\frac{\lambda}{\mu})}(e^{-\mu x} - e^{-\mu(n-\frac{\lambda}{\mu})x}) = e^{-\mu x}(1 + \frac{(\frac{\lambda}{\mu})^n p_0}{n!(1-\rho)}\frac{1-e^{-\mu(n-1-\frac{\lambda}{\mu})x}}{(n-1-\frac{\lambda}{\mu})}) \end{aligned} \tag{32}$$

Therefore the distribution function can be presented as

$$F_s(x) = 1 - P(S > x) \tag{33}$$

7.3 | Overview of the PDSonQueue approach

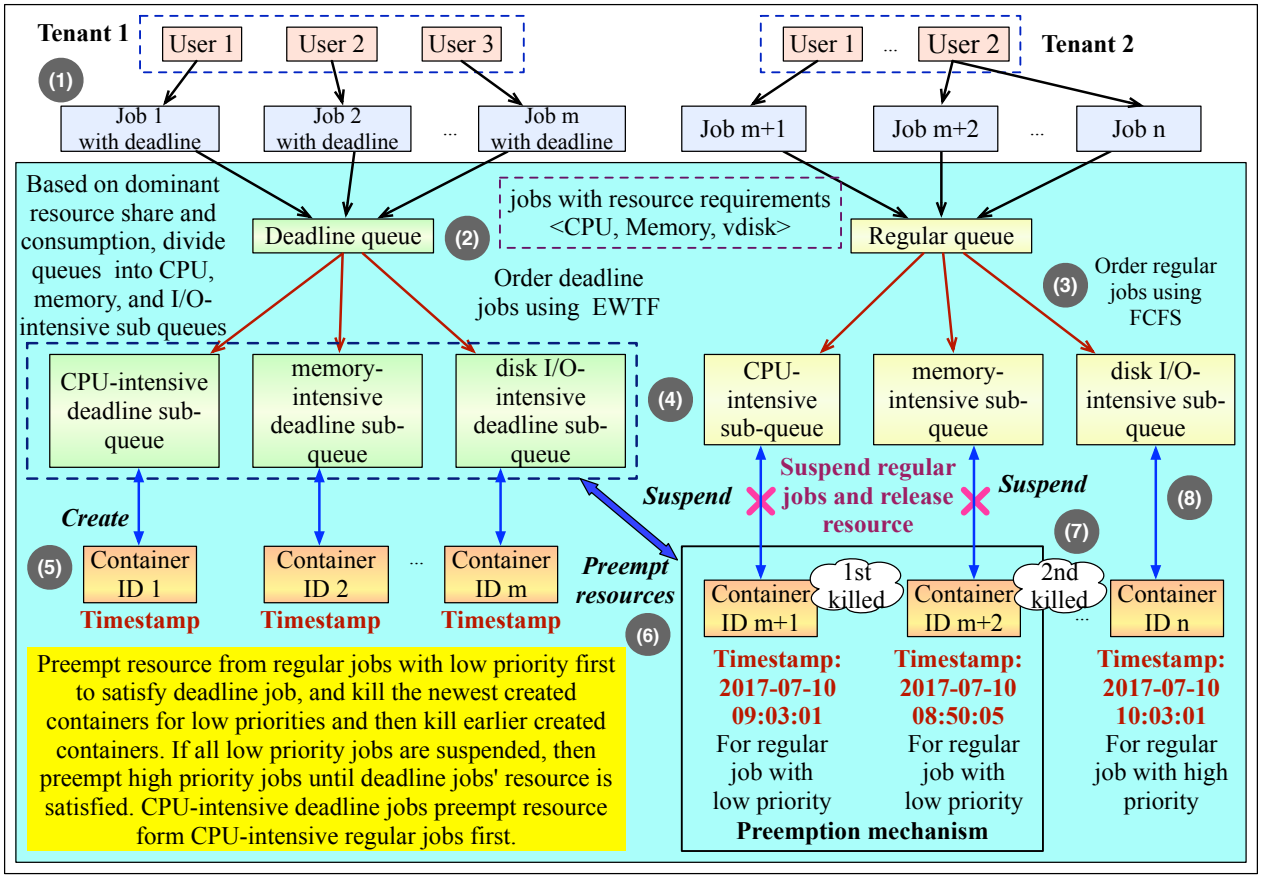

FIGURE 7 Deadline constrained scheduling model based on M/M/n queuing model

Preemptive scheduling heuristics are intended to judiciously accept, schedule and cancel the real-time services when necessary to maximise a system's QoS performance. Complimenting the previous non-preemptive algorithm, real-time jobs are scheduled preemptively with the objective of maximising the number of jobs accomplished before their deadlines and improving the efficiency of jobs. Traditionally, job killing is a simple way to release preemption. However, killed jobs cannot be resumed and have to be relaunched. Most cluster schedulers use this approach due to its simplicity[45]. Compared with this kind of kill-based preemption such as default Capacity Scheduler, our PDSonQueue uses advanced container-based preemption mechanism.

A container provides isolated namespaces for applications running inside and forms a resource accounting and allocation unit. Linux uses control groups (Cgroups) to precisely control the resource allocation to a container. Not only priorities can be set to reflect the relative importance of containers, but also hard resource limits guarantee that containers consume resources no more than a predefined upper bound even there are available resources in the system[45]. Containers can be created directly without the standardised templates and the containers that each job needs are customised by job owners on the scheduling level[76]. Each job is composed of a sequence of sub tasks, which would require the creation and utilisation of containers in order to interface efficiently between systems. Containers will apply for multidimensional resource (CPU, memory, disks, network bandwidth, etc.) from computing nodes.

The original YARN supports kill-based preemption only. However, in real system implementation, we modify our scheduler to support a suspend-based preemption. In our preemption mechanism, when there are not enough available resources, we suspend regular, low priority jobs to preempt resources, kill the corresponding containers, and release resources to the system. We introduce a "suspending queue" in our scheduling framework, which is used to put our suspended jobs. When these jobs are suspended, we put them in the suspending queue and change the state of these jobs from active to sleep. When there are more available resources, these "sleep" jobs can be resumed again.

Our PDSonQueue includes a preemption mechanism to preempt resources for deadline constrained jobs when available resources are insufficient, and a queuing model to estimate jobs' service time and waiting time, as shown in Figure 7. Users submit jobs to the cloud cluster (1). Jobs specify their key resource requirements: <CPU, memory, disk I/O> and QoS. There are 2 types of jobs: *deadline jobs* and *regular jobs*. Regular jobs do not have deadline requirements and are divided into low priority and high priority jobs. Correspondingly, 2 queues are built in PDSonQueue: deadline queue (2) sorts deadline jobs and regular queue (3) sorts regular jobs. According to different dominant resource consumption, PDSonQueue divides each queue into three sub-queues (4): CPU-intensive, memory-intensive and I/O-intensive sub-queues. Noted that, calculating the fraction of the job's required resources to the total resource, the largest one is selected as the dominant share in order to determine this job's type[24]. If a job's dominant resource share is both CPU and memory, so the job is both CPU and memory-intensive. According to the dominant resource share principle proposed in[24], putting the job in either CPU-intensive sub-queue or memory-intensive sub-queue can get the same results and does not influence scheduling efforts. For example, a CPU-intensive deadline job should be put into the CPU-intensive deadline sub-queue. We use EMWTF to sort deadline jobs. The deadline jobs with earliest waiting time should be run first. The regular jobs are sorted according to "First Come First Serve (FCFS)". When a job applies for resources successfully, a container is created with a timestamp (5). If deadline jobs cannot wait to get enough resources during their waiting time, they will preempt resources from regular, low priority jobs (6). Low priority jobs will be suspended, the corresponding containers will be killed and the resources will be released until there are enough resources to support the deadline jobs (7). When all low priority jobs are suspended but there are still insufficient resources, high priority regular jobs will be preempted (8). Our preemptive strategy chooses the latest served jobs to preempt and kills the containers based on timestamps and dominant share resource profile. Latest created containers with the same dominant share as a deadline job will be killed first.

7.4 | Pseudo Code of PDSonQueue

Algorithm 5 shows the pseudo code of PDSonQueue. When new job_m enters a queue (*Lines:2-3*), we estimate Job_m's waiting time W_m^{wai} and sojourn time W_m^{soj} based on M/M/n probability statistical model (*Line:4*). If job_m is deadline constrained, it is put in the deadline queue (*Lines:6-7*), otherwise it is put in the regular queue. We sort the deadline queue and the deadline constrained job_i which has the smallest waiting time W_i^{wai} should be processed first (*Line:8*). Through calculating the dominant share of each job $Domjob_m$ (*Line:5*), we classify deadline queue into sub-queues: CPU-, memory- and I/O-intensive and put the job to the corresponding sub queue (*Line:10*). Within its waiting time W_i^{wai}, it needs to successfully apply for the required resource Dem_i from the cloud system. If job_m is a regular job, it will be put in a regular queue based FIFO. Then the regular queue will also be classified into sub-queues as the deadline queue (*Lines:10-15*)

When the system's available resource Res_{rem} is smaller than job's required resource Dem_i (*Line:18*) and job's waiting time W_i^{wai} is smaller than resource released time, which is the remaining service time of next job being finished (*Line:19*), the system will preempt resources from regular, low priority jobs until Res_{rem} is larger than Dem_i (*Line:20*). We first preempt the resource from the jobs which have the same kind of dominant share as that of job_i. The running jobs in the regular sub-queue $queue_{reg}^{inten[*]}$ are sorted in an ascending order of their current running time. The newest, running job job_{newlo} from $queue_{reg}^{inten[*]}$, should be suspended and its resources Dem_{newlo} are released to the system. After adding Dem_{newlo}, if Res_{rem} is still smaller than the job_i's required resource Dem_i, we will suspend the second newest, job_{new2lo} to be run from $queue_{reg}^{inten[*]}$. We add the released resource Dem_{new2lo} to the cloud, and update Res_{rem} again. The preemption process iterates until Res_{rem} is larger than Dem_i. If all the low priority jobs from $queue_{reg}^{inten[*]}$ are suspended, but the updating Res_{rem} is still smaller than Dem_i, then the latest low priority jobs being run from other regular queues will be suspended to satisfy job_i's required resource Dem_i. If all the low priority jobs from different sub queues are suspended and Res_{rem} is still smaller than Dem_i, high priority jobs from $queue_{reg}^{inten[*]}$ are suspended first and then the latest high priority jobs from other queues are suspended secondly. When Res_{rem} being run is not smaller than Dem_i, the system allocates Dem_i amount resource to job_i (*Line:20*). The iteration of preempting resource and resource allocation should be accomplished in W_i^{wai}. Otherwise, the preemption process fails and job_i is unsuccessful to process.

Algorithm 5 PDSonQueue Scheduler

1: **// Scheduling phase**
2: **while** (jobs $\notin \phi$) **do**
3: record job parameter $job_i = \{Dem_i, t_i^{arr}, t_i^{dea}\}$ or $job_i = \{Dem_i, t_i^{arr}, p_i\}$
4: estimate job waiting time W_i^{wai} and sojourn time W_i^{soj}
5: calculate the dominant share of each job: $Domjob_i$
6: **if** (t_i^{dea}) **then** /***deadline job***/
7: put job_i into the deadline queue $queue_{dea}$
8: **sort**($queue_{dea}$, into an ascending order of W^{wai})
9: extract job_i from $queue_{dea}$ into $queue_{dea}^{inten[*]}$ based on $Domjob_i$, $inten[] = \{cpu, ram, io\}$
10: **else**
11: put job_i into the regular queue $queue_{reg}$
12: **sort** $queue_{reg}$ in FIFO principle
13: divide job_i from $queue_{reg}$ into $queue_{reg}^{inten[*]}$ based on $Domjob_i$, $inten[] = \{cpu, ram, io\}$
14: **end if**
15: **end while**
16: **// Preemption phase**
17: **if** (job_i is a deadline job) **then**
18: **if** ($Dem_i > Res_{rem}$) **then**
19: **if** ($W_i^{wai} <$ Resource released time) **then**
20: suspend the newest running low priority regular jobs job_{newlo} in the $queue_{reg}^{inten[*]}$ ($*$ is calculated based on $Domjob_i$) until the system has Dem_i resource available, and allocate required resource Dem_i to job_i
21: **else**
22: **if** ($Dem_i > (Res_{rem} + Dem_{nexfin})$) **then**
23: suspend the newest running low priority regular jobs job_{newlo} in $queue_{reg}^{inten[*]}$ until the system has Dem_i resource available, and allocate required resource Dem_i to job_i
24: **else**
25: job_i waits until next job being finished job_{nexfin} is done and job_{nexfin}'s resource is released; after releasing resource, allocate required resource Dem_i to job_i
26: **end if**
27: **end if**
28: **else**
29: allocate required resource Dem_i to job_i
30: **end if**
31: **else** /***job_i is a regular job***/
32: **if** ($Dem_i > Res_{rem}$) **then**
33: job_i waits until other resource released; after releasing enough resource, allocate required resource Dem_i to job_i
34: **else**
35: allocate required resource Dem_i to job_i
36: **end if**
37: **end if**
38: preempt resource and allocate required resource Dem_i to job_i
39: **return** job_i begins to run

When Res_{rem} is smaller than Dem_i and W_i^{wai} is smaller than resource released time, if requried resources, Dem_i is larger than next be finished job's (job_{nexfin}'s) resource amount, Dem_{nexfin}, plus current system's available resources, Res_{rem}, preemption will start and the preemption process used is the same as stated above (*Lines:22-23*). If Dem_i is smaller than next being finished job's (job_{nexfin}'s) resource amount, Dem_{nexfin}, plus current system's available resources, Res_e, job_i waits until $job_{nextfin}$ is accomplished and then gains enough resources (*Lines:24-26*). If Dem_i is smaller than Res_{rem}, the system does not preempt jobs

for their resources (*Lines:28-29*). If job_i is a regular job, the system still does not need to preempt for resources. If there is not enough available resources, they simply wait (*Lines:31-35*). Preemption process for n jobs has the time complexity of $O(n)$.

8 | INTEGRATION OF INDIVIDUAL SCHEDULERS

From the perspective of tenants, our scheduling framework – ***3DSF satisfies their deadline based QoS requirements and puts their tasks near required data***, and from the perspective of service providers, it ***completes more jobs with higher resource utilisation and reduces their total operation costs***. Our ***3DSF*** addresses the issues on how to satisfy different tenants' QoS requirements, without sacrificing performance and other important elements such as SLA accordance and the data locality issue.

Our ***3DSF*** is a generic framework, without platform limitations. Our ***3DSF*** uses an adaptive suite to choose the corresponding scheduling strategy that satisfies the differing profiles of jobs that are submitted with factors such as the resource requirements, the specific resource allocation, customised data locality and other special QoS. The purpose of our ***3DSF*** is to: 1) maximise resource utilisation, and reduce the deadline-based SLA violations; 2) improve system throughput, data locality and network bandwidth utilisation; and 3) reduce the completion time of jobs and minimise system's overhead. In ***3DSF***, there are 5 primary modules. *Module 1* focuses on dynamic resource allocation to improve system's throughput. Note that deadline jobs usually have higher priority with a subsequently higher capacity to preempt resources from other jobs, so *module 2* and *module 3* are merged together to realise a better QoS-based scheduler, including a performance prediction model. *Module 4* answers the problem how to determine when to move computation vs when to move data and enhance data locality, while accelerating the system's throughput and performance. *Module 5* is an adaptive suite to realise the comprehensive scheduling framework, which can invoke corresponding scheduling modules according to different scenarios, aiming at adapting to different scheduling requirements and environments. A scheduling framework that can handle all the aforementioned issues simultaneously would

have immense usability. Our ***3DSF*** works for multi-tenants in clouds, especially for big data and large applications. As when the data size is too small, the overhead of our ***3DSF*** could be high. In addition, our work does not consider the privacy and security issue and the relevance of data of multi-tenants and assume that jobs from multi-tenants are independent, which are the possible shortcomings of our work.

8.1 | Architecture of our scheduling framework – 3DSF

Figure 8 shows the architecture of ***3DSF***, which is the detailed implementation of Figure 1. On the top layer, tenants submit different application requests with varying QoS requirements (1). Our framework automatically analyses jobs (application requests)'s information to get the jobs' priority, deadline, preemption possibility and resources requirements (2). Subsequently, we are then able to build prediction training models for this framework (3). The prediction models include a neural network and an M/M/n queuing model (4). The neural network uses a two hidden layers Multilayer Perceptron (MLP) and is employed at the task scheduling level as a part of the data locality aware scheduler. It predicts tasks' remaining execution time and resource releasing time (5). The queuing model is employed at the job scheduling level and is a part of the deadline constrained scheduler. It uses probability theory and statistical analysis to predict the execution time and waiting time of the jobs (6). To improve the accuracy of the prediction models, we use historical data to train the prediction models in order to obtain more accurate predictions. Our framework uses an adaptive suite (*Module 5*), to determine the submitted jobs going through the deadline constrained scheduler (*Modules 2&3*) and/or the data locality aware scheduler (*Module 4*) (7). If jobs have deadline requirements, the jobs will go through the deadline constrained scheduler. Deadline jobs have a higher priority and can preempt resources from regular (non-deadline) jobs, with the aim of guaranteeing that deadline jobs are completed within their deadlines (8). At the middle layer of a distributed environment, jobs are divided into tasks and these tasks are processed with their required data in parallel. If the jobs are data-intensive, and the required data is large in magnitude, we need to use the data locality aware scheduler (*Module 4*) to enhance data locality performance (9). If there is no need to employ the data locality aware scheduler, we will subsequently skip *Module 4* and directly place tasks into corresponding queues in order to wait for resource allocation. *Module 4* uses a judgment mechanism to determine when it is best to move data and when it is best to move computation, in order to improve data locality performance. After going through *Modules 2&3* and *Module 4*, tasks are put into corresponding intensive queues where they wait to get their relevant resources from the cloud system. At the bottom layer, our scheduling framework considers < CPU, memory, disk I/O> to fairly allocate resources to the tasks of *Module 1* (10). *Module 1* uses Linux containers to bundle the required resources and flexibly provides containers to the tasks. In *Module 1*, *Cgroup* compulsively controls resource utilsation.

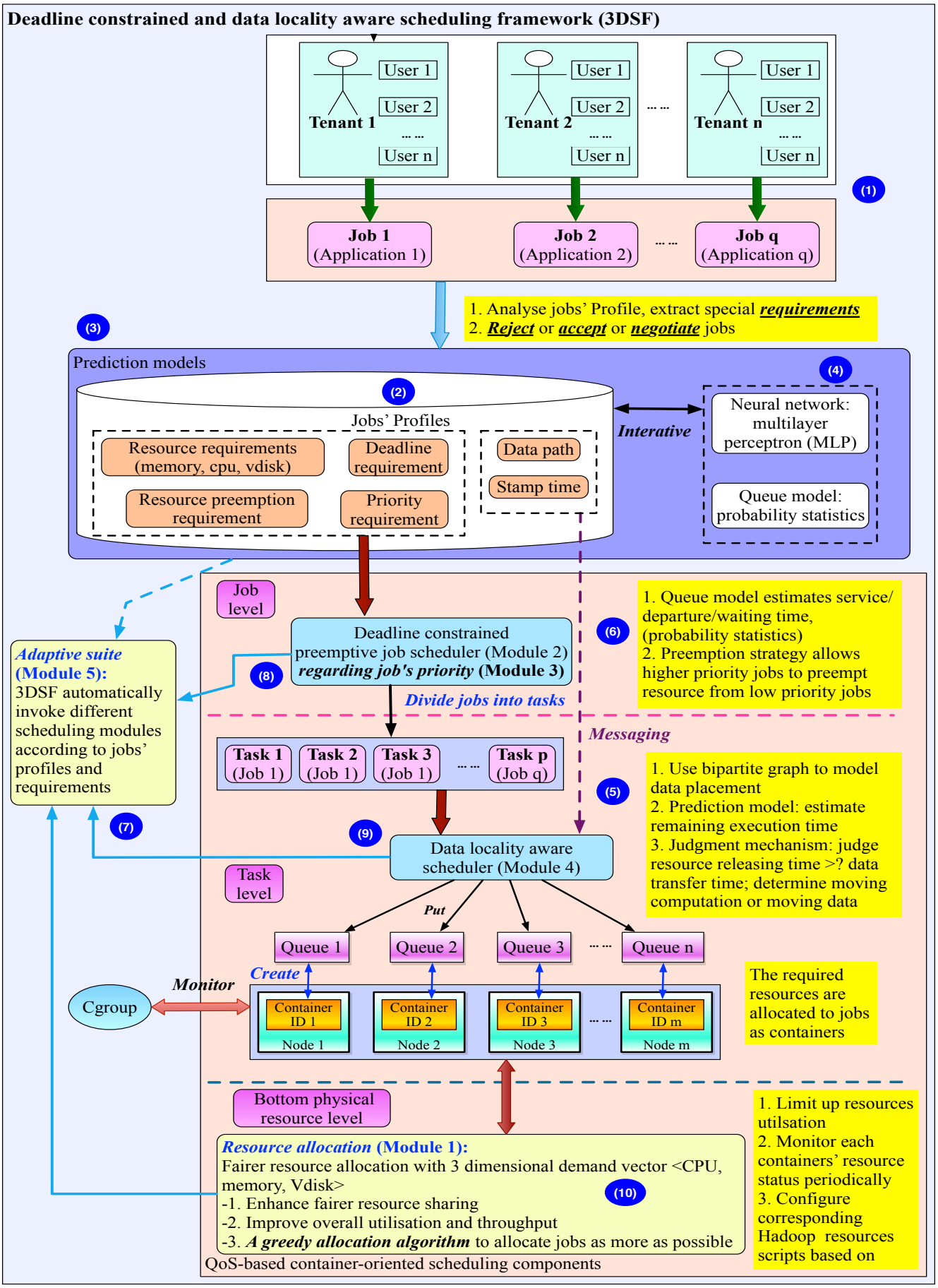

FIGURE 8 Architecture of *3DSF*

8.2 | Pseudo Code of 3DSF

Algorithm 6 3DSF Scheduling Framework

Input: A set J of jobs with different resource requirements and a set S of nodes with heterogeneous performance
Output: **3DSF** scheduling for the tasks of J

```
1:  while (jobs are submitted to the cloud) do
2:      if (jobs J are deadline jobs) then
3:          // Deadline constrained scheduler
4:          Invoke Algorithm 5 to schedule jobs J
5:      end if
6:  end while
7:  // Job partition
8:  while (jobs J are divided into tasks T) do
9:      if (tasks T require large amount of data to process) then
10:         // Data locality aware scheduler
11:         for (each task t of T) do
12:             Invoke Algorithm 2 to determine moving data or moving computation for task t, and then locate task t to
    appropriate node s based on the judgment mechanism
13:         end for
14:     end if
15:     Create a container con_t for task t, and con_t has task t's required resources
16:     // Resource allocation strategy
17:     Invoke Algorithm 1 to decide which job should be run next
18: end while
```

Algorithm 6 describes the basic approach of ***3DSF***. When jobs are submitted, the profiles of the jobs and the cloud system are initialised (*Line:1*). If the jobs that have been submitted to process have deadlines, ***3DSF*** invokes deadline constrained scheduler – PDSonQueue[65] to schedule the jobs according to their deadlines (*Lines:2-4*). After partition, the job j is divided into tasks (*Lines:7-8*). If tasks T require large amount of data, with the aim of enhancing tasks' data locality (*Line:9*), the data locality aware scheduler – DLAforBT[64] is invoked to locate tasks to their corresponding nodes to improve data locality (*Lines:10-12*). Otherwise, if tasks T are small, the optional data locality aware scheduler – DLAforBT[64] will be skipped and tasks T directly wait for resource allocation. A corresponding container con_t with a timestamp is created for task t (*Line:15*). For resource allocation, we use a greedy strategy – improved DRF algorithm[24] to enhance fairer resource sharing and to decide which tenant's job should be run (*Lines:16-17*).

Although the complexity reaches $O(n^3)$, compared to jobs execution, the overhead of our scheduling framework is negligible. In our empirical experiments, we can clearly see that the overhead of scheduling process is very small, comparing with running jobs. Thus, the scheduling overhead is not an issue.

9 | EXPERIMENTS

To demonstrate our ***3DSF***'s high efficiency performance, we have implemented a version of our ***3DSF*** for YARN. YARN[1] is an open source implementation to provide resource management and a central platform to deliver consistent operations, security, and data governance tools across Hadoop clusters. YARN is also a resource management system which is in charge of resource management and scheduling. It supports dynamic resource allocation according to the actual demands of tenants, with fine-grained allocation being based on different types of resource consumption, and resource isolation. YARN has gained significant popularity as a platform for large scale data processing applications. YARN as next generation of Hadoop is a successful, comprehensive and widely used commercial product in IT industry, which is one of the most popular cloud platforms. Due to technology's development and increasing requirements of customers, YARN is continually updated. We choose the recent

version of YARN. Our experiments focus on different performance metrics under different types of workloads. To evaluate the superiority of our scheduling framework, we choose the recent version of YARN's scheduler as a benchmark to compare with our

3DSF. Some other scheduling frameworks such as[23], Mesos[2], and[63] also use Hadoop as a benchmark to make a comparison. To evaluate our scheduling framework's performance, we employ 5 metrics: resource utilisation, deadline-based QoS, data locality, throughput, and completion time to evaluate different types of MapReduce-based workloads. Our experiments compare our new scheduling framework's performance for both single type of workload and mixed workloads to YARN Capacity Scheduler[7]. Our experiments contain 2 groups of applications: four exemplar benchmark applications and four real applications, each with either a single type of workload or with mixed workloads. This validates our framework's performance for practical applications. For each real application, a corresponding 40GB data file is selected as input, respectively. We have 4 real applications. Thus, we process about 200GB data at the same time in our cloud.

Capacity Scheduler is proposed by Yahoo. It supports parallel operations of multi tasks and dynamically allocates resources to increase the efficiency of tasks' execution[7]. The fundamental basics of Capacity Scheduler are around how queues are laid out and resources are allocated to them. When a task is submitted, it is put into a queue. Each queue gets some resources based on configuration to process operations. Tasks can be operated according to their different priorities. A high-level priority task will be executed first, but Capacity Scheduler does not support preempting priority. A comprehensive set of limits is provided to prevent a single job, user or queue from monopolising the resources of the queue or the cluster as a whole to ensure that the system is not overwhelmed by too many tasks or jobs.

Our experiments aim to evaluating the entire scheduling framework's performance when different types of workloads come. For each individual scheduling component (Sections 5, 6 and 7), we have also conducted a large number of experiments to evaluate each scheduler's superiority in our previous works. For instance, our improved DRF algorithm presented in Section 5 is described in our work[24] for details. In work[24], we did a large number of experiments to compare our improved DRF with the naïve DRF. The experimental results indicated that our algorithm's performance is better than the naïve DRF. We also tested I/O usage restriction control. On the contrary, the naïve DRF was proposed in work[35] and was only compared with slot-based and CPU-based fair scheduling algorithm. Similarly, more experiments about DLAforBT presented in Section 6 were described in our work[64] and more experiments about PDSonQueue showed in Section 7 were described in our work[65].

Deadline is our major consideration, so the key QoS constraint means the proportion of completed deadline jobs in all deadline jobs. The deadlines of jobs are followed by normal distribution. Before experiments, we run all kinds of applications on our cloud, to gain average execution time for each type of application, marked as μ. According to Pauta Criterion (3σ Criterion), we set each type of application's deadline $\in (\mu - 3\sigma, \mu + 3\sigma)$ and the probability of deadline $\in (\mu - 3\sigma, \mu + 3\sigma)$ is 0.9973[77].

Locality rate means the ratio of local data read over all data read, which is one of our major considerations. Our experimental applications include long-term jobs and short-term jobs. Only using the number of jobs to measure the *throughput* metric would reduce the accuracy. Hence, we use "job unit" to generalise different jobs and the normalised job unit is 100ms, so $Throughput = \frac{\sum_{job_i \in completed\ \ jobs} W_i^{soj}/job\ unit}{\text{Completion time}}$, where W_i^{soj} is job i's sojourn time.

Our cloud cluster contains 5 machines each with 16GB of RAM, 2.9 GHz 8 cores Intel Xeon Processors, 3 1TB disk drives, running Hadoop YARN 2.6.0 on an Ubuntu server. The results from the experiments have been evaluated for 30 times and the figures show the average results based on the evaluation.

Before we performed the experiments in this work, we conducted a number of pilot experiments. In the pilot experiments, we set up different job parameters and job sizes to determine the final experiment design. We found that when the cloud computing capacity is fixed, the job size increases and thus the total completion time also increases but the throughout is almost the same. For example, in the pilot experiments, even if the job size was set up as 60,000 or 90,000, we still can get the similar conclusions. A scheduling framework's performance is not related to the job size, but related to the cloud computing capacity and inherent scheduling algorithms. It is very important to choose the suitable job size to conduct the experiments, which could avoid the cloud system being over- or under-loaded. An over- or under-loaded cloud system will have a negative effect on the experiment effort and effectiveness to evaluate our scheduling framework's performance. Therefore, according to our cloud cluster's capacity and cloud's scalability and elasticity, we finally choose 3 groups of jobs (6,000, 18,000, and 30,000) to conduct our final experiments.

9.1 | Exemplar Benchmark Applications and Evaluation Results

We selected four of Hadoop's classical benchmarks as follows: (1) **TeraSort** samples the input data and uses map/reduce to sort the data into a total order, which is implemented as a standard MapReduce sort with a custom partitioner that uses a sorted list of $(n-1)$ sampled keys that define the key reduce. (2) **Pi estimator** is a pure ***CPU-intensive*** application that employs a Monte

Carlo method to estimate the value of Pi. All map tasks are independent and a single reduce task gathers very little data from map tasks. (3) **Malloc** is a classical ***memory-intensive*** task to allocate unused space for an object whose size in bytes is specified by size and whose value is unspecified. (4) **Read/Write file** is a simple ***I/O-intensive*** task that reads and writes files repeatedly and continuously. Reading frequency equals to writing frequency.

(1) Deadline-based QoS, throughput, locality rate, completion time and resource utilisation when running TeraSort jobs:

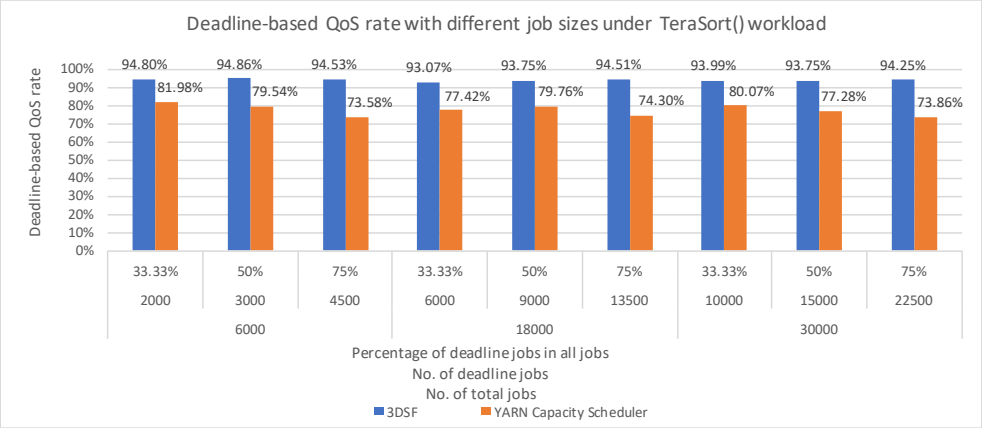

FIGURE 9 Deadline-based QoS rate on TeraSort() jobs workload

The job sizes are 6,000, 18,000 and 30,000, respectively, with deadline jobs occupying 33.33%, 50% and 75% of the total jobs. In Figure 9, our 3DSF's QoS achievement is 94.17% on average. Whatever the fraction of deadline jobs in total jobs and the total number of jobs are changed, our 3DSF QoS rate is quite flat and steady. However, compared with our 3DSF, Capacity Scheduler's QoS is 77.53% on average, which is lower by approximately 17%. With deadline jobs occupying 75% of the total jobs, we can observe that Capacity Scheduler's QoS is much lower than when deadline jobs occupying 33.33% and 50% in total jobs. For example, in the first 3 columns of Figure 9 (the total number of jobs is 6,000), when the fraction of deadline jobs in total jobs is 33.33%, Capacity Scheduler's QoS is 81.98%, which is 8.4% higher than that of when the fraction of deadline jobs in total jobs is 75%. As the amount of deadline jobs increases, the QoS rate of Capacity Scheduler suffers as a result.

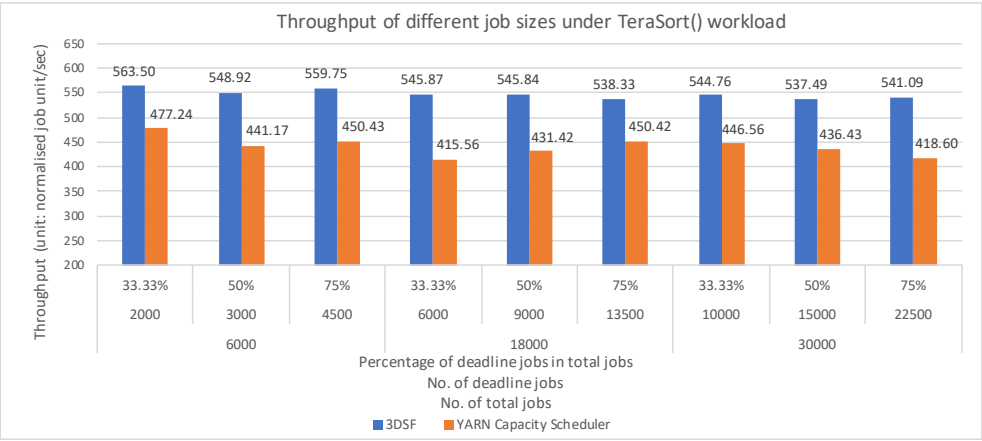

FIGURE 10 Throughput on TeraSort() jobs workload

Figure 10 shows the throughput results of running TeraSort() jobs. When the number of deadline jobs increases, the throughput decreases. When more deadline jobs enter the system, to meet these deadline jobs being finished on time, more regular jobs will be suspended, subsequently decreasing the throughput. The average throughput on a TeraSort() workload using Capacity Scheduler is only 440.87 job units/s. However, the average throughput of running TeraSort() workload using 3DSF is 547.28 job units/s, which is an improvement of 24.14%. When the total number of jobs increases, the mean throughout increases slightly. For instance, when the total number of jobs is 18,000, the throughput of our 3DSF is 543.34 job units/s and the throughput of Capacity Scheduler is 432.46 job units/s. When the total number of jobs is 30,000, the throughput of our 3DSF is 557.39 job units/s and the throughput of Capacity Scheduler is 456.28 job units/s, improved by about 20 job units/s.

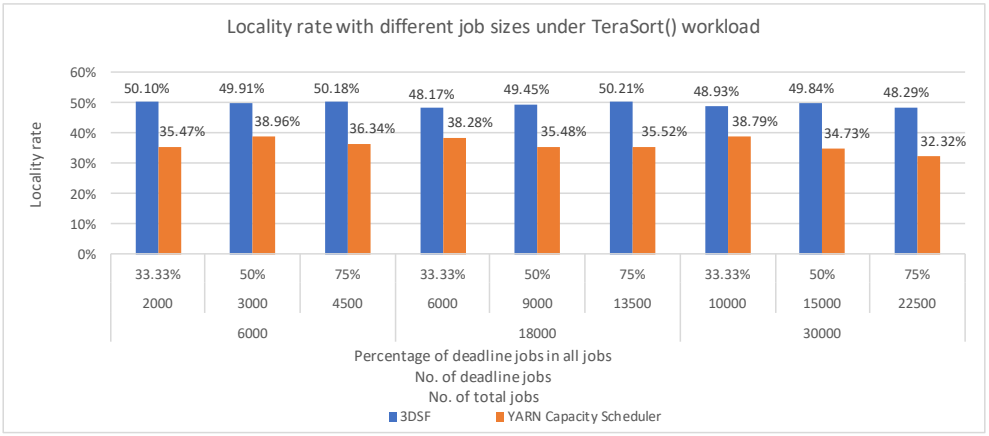

FIGURE 11 Locality rate on TeraSort() jobs workload

Figure 11 presents the locality rate. Our 3DSF's average locality rate achieves 49.45% while the YARN Capacity Scheduler's mean locality rate is only 36.21%. We can see that our scheduling framework has improved upon the locality rate by 13.24%. Our 3DSF not only emphasises the data locality issue, but also considers the tradeoff of jobs.

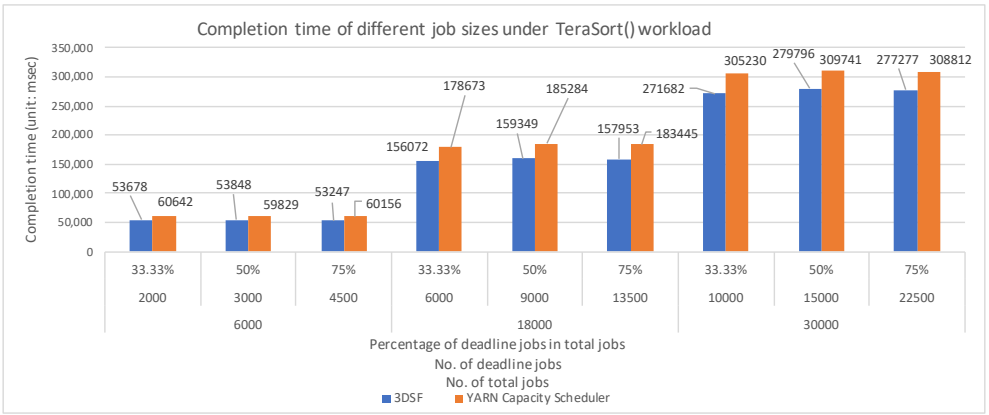

FIGURE 12 Completion time on TeraSort() jobs workload

Figure 12 presents the completion time on TeraSort() jobs workload. We clearly see that when the total number of jobs is 6,000, the completion time is smaller than the completion time when the total number of jobs is 18,000 and 30,000, respectively. The average completion time of our 3DSF is 53,591 ms, which is 10.99% faster compared with Capacity Scheduler which had an average completion time of 60,209 ms. When the total number of jobs is 18,000 and 30,000, we can get similar results. On average, 3DSF can reduce jobs' completion time by over 11.60%.

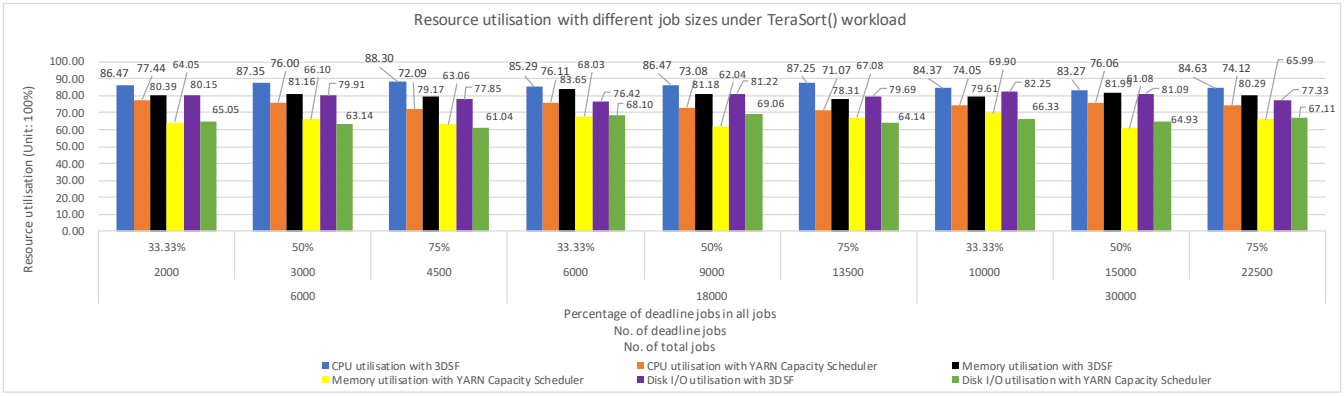

FIGURE 13 Resource utilisation on TeraSort() jobs workload

Figure 13 presents the average resource utilisation when running TeraSort() jobs. Using our 3DSF, the CPU usage rate reaches 85.93%, memory utilisation is 80.64% and I/O utilisation reaches 79.55% on average. Using YARN Capacity Scheduler, CPU usage rate only reaches 74.45%, memory utilisation reaches 65.26% and I/O utilisation gets to 65.43% on average. Compared with Capacity Scheduler, our 3DSF can improve the CPU utilisation by 11.49%, the memory utilisation by 15.38% and the I/O utilisation by 14%. TeraSort() is a small application, so the CPU utilisation improves less than the other 2 resources.

(2) Deadline-based QoS, throughput, locality rate, completion time and resource utilisation when running mixed workload (Pi estimator, Malloc and Read/Write file):

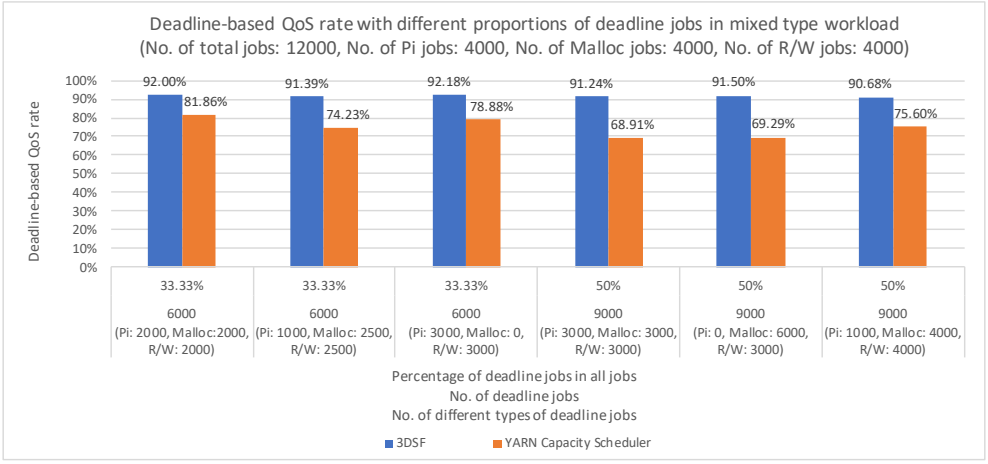

FIGURE 14 Deadline-based QoS rate on a mixed workload

We define a series of different job combinations, the first combination based on setting deadline jobs to occupy 33.33% and 50% of the total jobs and changing the number of deadline Pi estimation jobs, Malloc jobs and read/write jobs. Figure 14 shows the QoS achievements with these different job combinations. When the fraction of deadline jobs is 33.33%, the average QoS rate of 3DSF is 91.85% without regarding proportion of the total workload that is composed of deadline jobs. The YARN Capacity Scheduler's QoS rate is 78.32%. Meanwhile, when deadline jobs occupy 50% of the total jobs, our framework's average QoS rate is 91.14% and the Capacity Scheduler's QoS rate is 71.26%. Our 3DSF's performance continually fluctuates and averages 91.50% whenever the number of deadline jobs changes, which is higher by 16.70% than Capacity Scheduler. The YARN Capacity Scheduler's QoS rate decreases by 7.06% when the number of deadline jobs changed from 6,000 to 9,000.

Figure 15 presents the throughput of running mixed types of jobs. Our 3DSF's average throughput is higher (591.95 job units/s) than that of the YARN Capacity Scheduler (492.29 job units/s), which gains 20.24% improvement. When the number of deadline

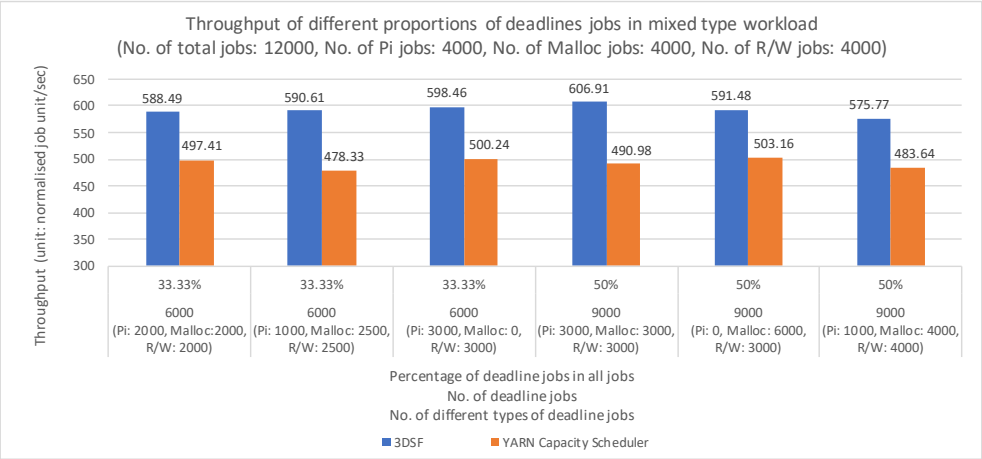

FIGURE 15 Throughput on a mixed workload

jobs are 9,000 and the number of Pi estimation jobs, Malloc jobs and read/write jobs are 1,000, 4,000, and 4,000 respectively (the last group in Figure 15), the throughput is slightly lower than other groups, for both 3SDF and Capacity Scheduler.

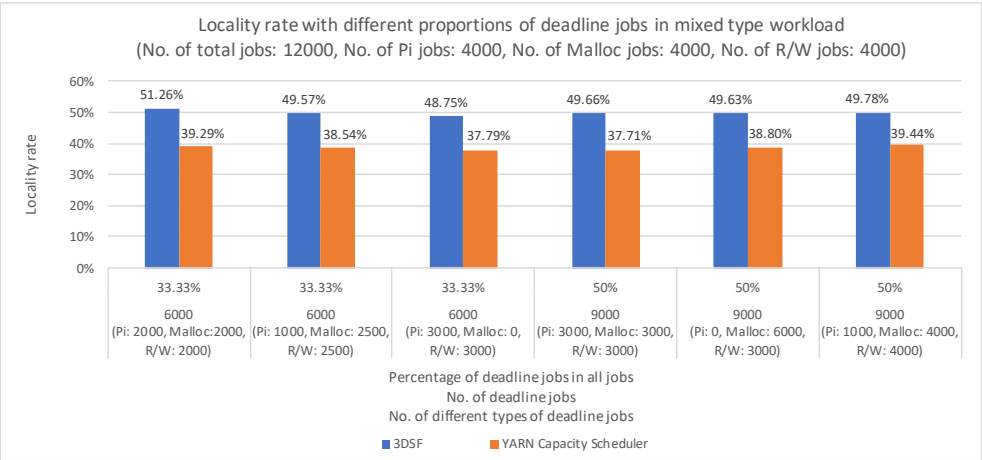

FIGURE 16 Locality rate on a mixed workload

Figure 16 presents the locality rate achieved with these different job combinations. The average locality rate for 3DSF is 49.78%. However, whenever the number of jobs or the combinations of jobs are changed, the YARN scheduler's mean locality rate is only 38.60%, which is 11.18% lower than 3DSF.

Figure 17 shows the completion times achieved by the running of mixed types of jobs. Since the total number of jobs is fixed as 18,000, the completion time does not fluctuate significantly. Using YARN Capacity Scheduler, the mean completion time is 212,331 ms. The average completion time of 3DSF is 177,040 ms, which can deteriorate by 16.61%.

Figure 18 shows the resource utilisation obtained when running mixed types of jobs. Using our 3DSF, average CPU utilisation achieves 91.17%. When the number of deadline jobs is 9,000 and the number of Pi estimation jobs, Malloc jobs and read/write jobs are 1,000, 4,000, and 4,000 respectively, the CPU utlisation of 3DSF is 89.21%, which is slightly lower than other groups. The reason for this phenomenon is that the throughput is lower as shown in Figure 15. The 3DSF's average memory utilisation is 80.45% with the I/O utilisation being 79.88%. Compared with our 3DSF, the default YARN Capacity Scheduler's average CPU utilisation is 77.03%, lower than 3DSF by 14.13%, with an average memory utilisation of 63.90%, lower than 3DSF by 16.55%, and an average I/O utilisation of 64.84%, lower than 3DSF by 15.04%, respectively.

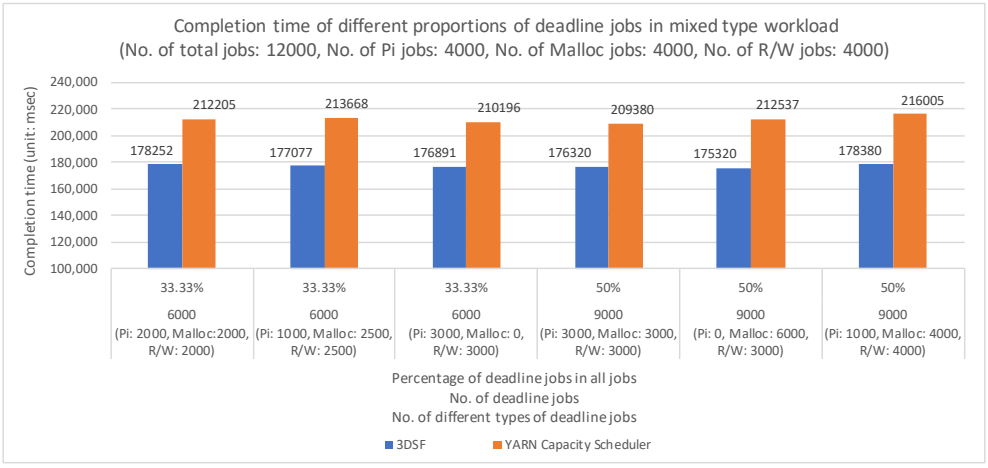

FIGURE 17 Completion time on a mixed workload

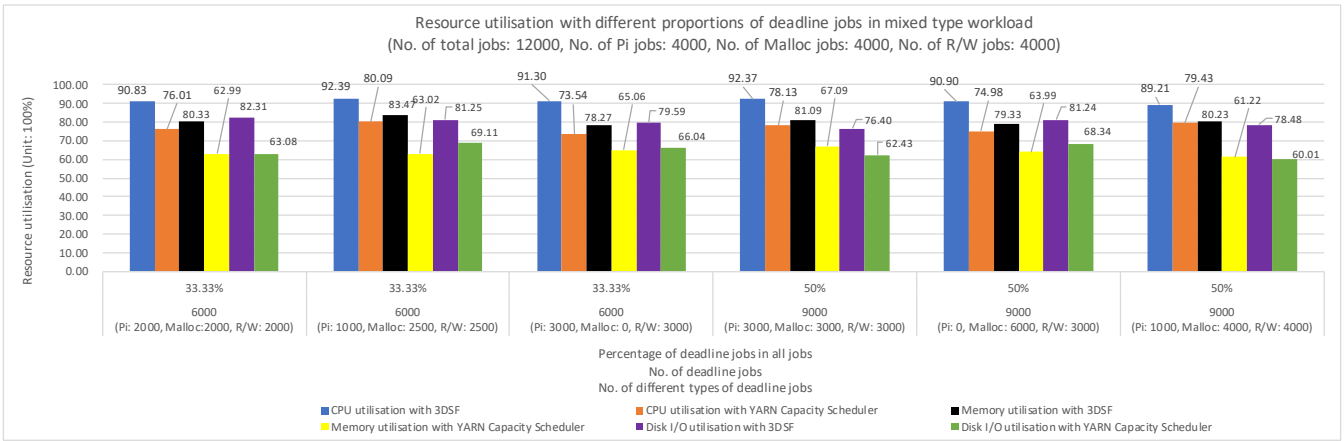

FIGURE 18 Resource utilisation on a mixed workload

9.2 | Real-world Applications and Evaluation Results

We evaluated our ***3DSF*** vs YARN's Capacity Scheduler performance using 4 real world applications.

Case 1: Enterprise telecommunication data analysis application: This data analysis application mines and analyses information from customer mobile call usage history, based on classifying and ordering user groups, customer behaviour and plans. This is a ***CPU-memory-I/O-intensive mixed type application***. A 40G data file is split into blocks, deployed on different nodes, duplicated data on disks and read into memory from disks. Then the application scans and indexes the data, and the classification and analysis process consumes CPU and multi-dimensional tables consume memory and disk I/O.

Case 2: Number plate image recognition: This License Plate Recognition System (LPRS) recognises a vehicle plate license from images with edge detection used to identify points in digital images with discontinuities. Edge detection calculates every pixel of an image, with complexity $O(n^2)$. A 40G image data file is loaded and read once from disk. This application is predominantly ***CPU-intensive***.

Case 3: Hadoop log file text search: This application tracks Hadoop's logs to search for error information using a simple lambda expression based on the "error" string, identifying a cluster's health status and weakness. Its complexity is low and it consumes little CPU resource. A 40G log file is buffered in memory, read and searched. This application is ***Memory-intensive***.

Case 4: Hadoop data migration: In a Hadoop cluster, the input file is split into one or more blocks stored in a set of DataNodes (running on commodity machines). When data volume is huge, tasks split from jobs are deployed on one node, and however the needed data may be stored on different nodes and even different racks. Thus the system needs to copy other nodes' data to this destination node. A 40G telecommunications data file is copied and transmitted among nodes. This application is ***I/O-intensive***.

(1) Deadline-based QoS, throughput, locality rate, completion time and resource utilisation when running case 1

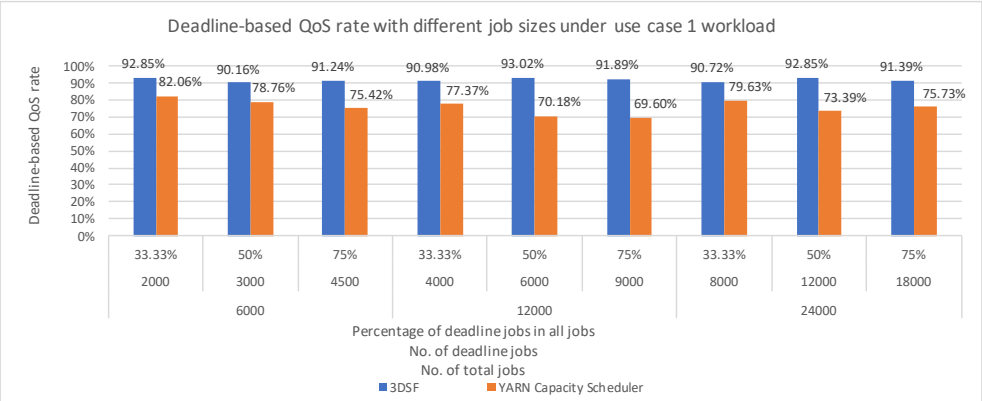

FIGURE 19 Deadline-based QoS rate on the workload of case 1

The job sizes are 6,000, 12,000 and 24,000 respectively, and deadline jobs occupy 33.33%, 50% and 75% of the total jobs. Figure 19 shows deadline-based QoS achievement rate when running case 1. Our 3DSF's average QoS rate is 91.68%. Capacity Scheduler's QoS rate is only 75.79%. Our 3DSF attains a deadline-based QoS rate that is 15.88% higher than Capacity Scheduler.

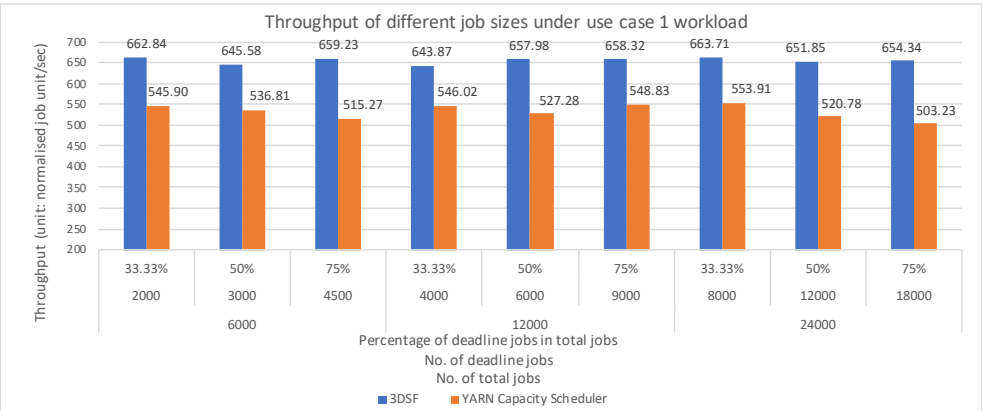

FIGURE 20 Throughput on the workload running case 1

Figure 20 presents the throughput results of running telecommunication application. Using Capacity Scheduler, the average throughput is 533.11 job units/s. Using 3DSF, the average throughput is 655.30 job units/s, which is higher than the YARN scheduler by 122.19 job units/s, achieving a 22.92% improvement over Capacity Scheduler. For larger applications, the throughput of both Capacity Scheduler and 3DSF are appreciably higher than the throughput of small exemplar benchmark applications.

Figure 21 shows the locality rate achieved when running case 1. 3DSF's average locality rate is 48.87%, and Capacity Scheduler's average locality rate is only 33.02%. 3DSF improves upon the YARN scheduler's locality rate by 15.85%.

Figure 22 presents the completion times under the case 1 workload. The more deadline jobs are submitted in the system, the higher the average completion time subsequently becomes. Using 3DSF, the average completion time when the total number of jobs is 6,000 is 1,083s. The average completion time when the total number of jobs is 12,000 is 2,221s, and the average completion time when the total number of jobs is 24,000 is 4,489s. Compared with YARN Capacity Scheduler, our 3DSF can reduce the completion time by 16.85% on average. Under the TeraSort() workload, 3DSF only reduces the completion time by 11.60%. As the size of the application increases, the efficiency gains caused by 3DSF will also increase.

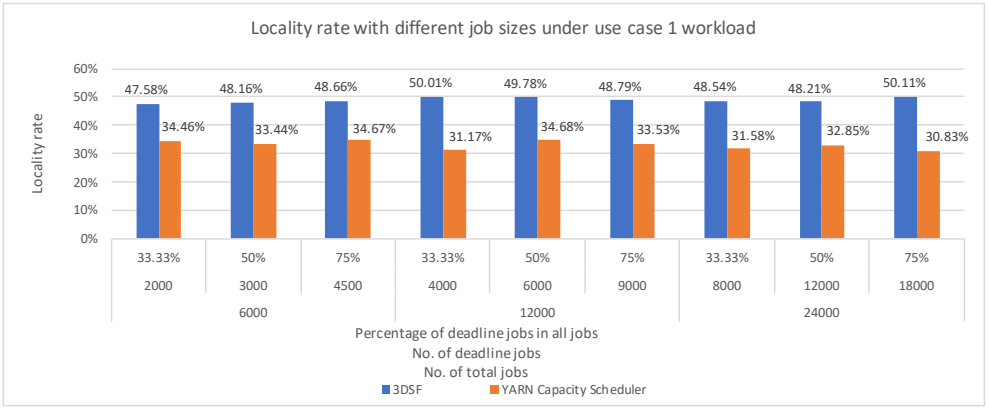

FIGURE 21 Locality rate on the workload running case 1

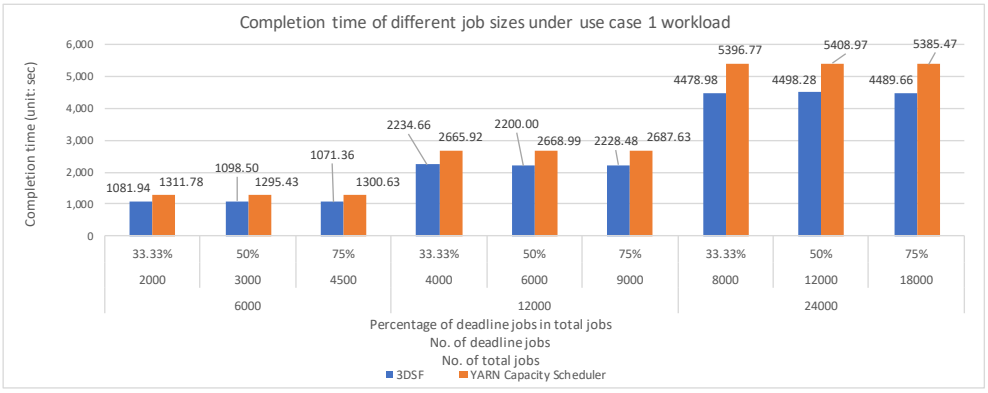

FIGURE 22 Completion time on the workload running case 1

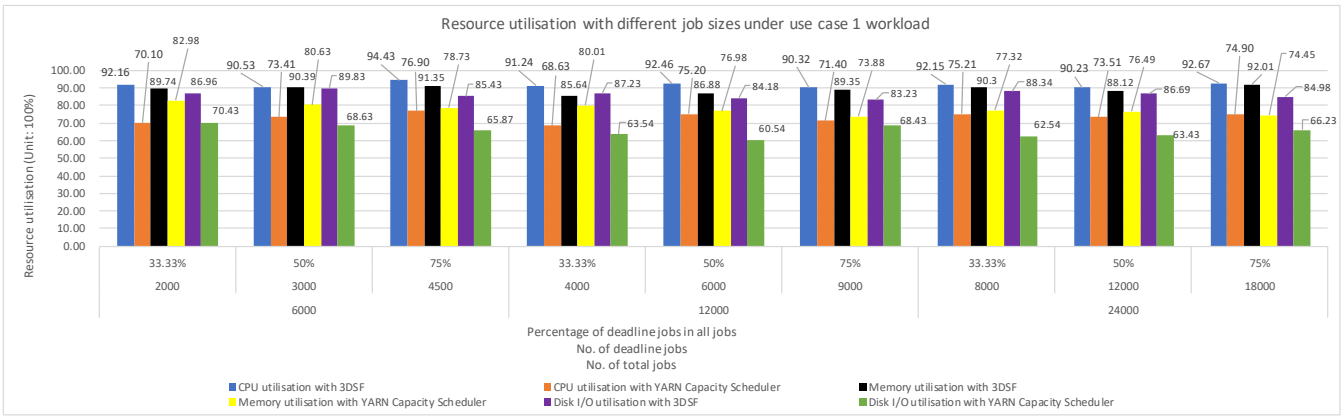

FIGURE 23 Resource utilisation on the workload running case 1

This application is of a mixed type workload, with all three kinds of resources being utilised to a relatively high level as shown in Figure 23. Using the 3DSF, CPU usage rate reaches 91.80%, memory utilisation reaches 89.31% and I/O utilisation reaches 86.32% on average. However, compared with 3DSF, the average CPU utilisation of YARN Capacity Scheduler is 73.25%, being lower by 18.54%, the average memory utilisation of Capacity Scheduler is 77.94% and is lower by 11.36%, and the average I/O utilisation of Capacity Scheduler is 65.52% and is lower by 20.80%. The performance of Capacity Scheduler deteriorates appreciably as the size of the application increases.

(2) Deadline-based QoS, throughput, locality rate, completion time and resource utilisation when running cases 2, 3 and 4 together

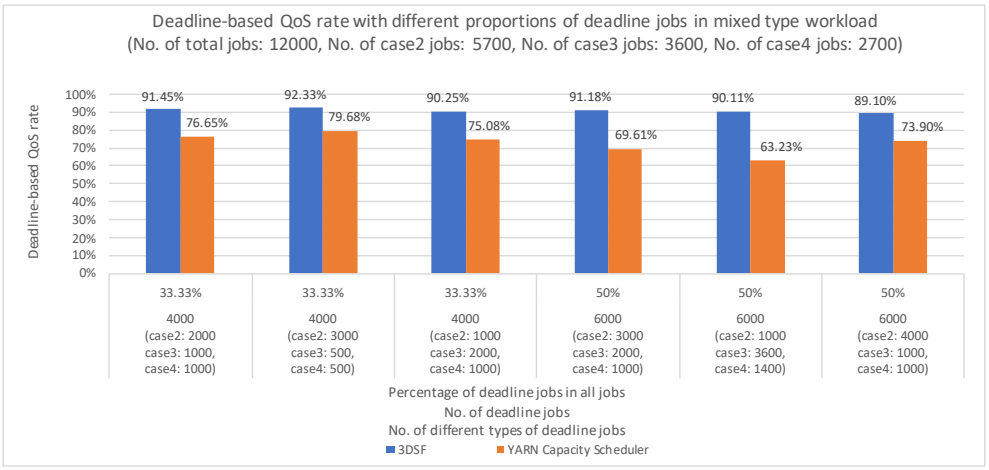

FIGURE 24 Deadline-based QoS rate on a mixed cases workload

There are 12,000 jobs, including 5,700 case 2 jobs, 3,600 case 3 jobs, and 2,700 case 4 jobs. In the experimental setting, deadline jobs occupy 33.33%, 50% of total jobs, respectively. There are 6 different group combinations. Group 1 has 4,000 deadline jobs including 2,000 case 2 jobs, 1,000 case 3 jobs and 1,000 case 4 jobs with other group combinations being presented in horizontal axis of Figure 24. Figure 24 presents the QoS rate, and our 3DSF's average QoS rate is 90.74% with Capacity Scheduler' averaging QoS rate of 73.03%, demonstrating that 3DSF maintains a 17.71% more consistent QoS rate. When deadline jobs occupy 33% of total jobs while using Capacity Scheduler, the average QoS rate is higher (77.14%) than the QoS rate obtained when deadline jobs occupy 50% of the total jobs (68.91%). When the number of deadline jobs is higher, the percent of deadline jobs that fail to be processed increases. However, our 3DSF's performance remains consistent even as the percentage of deadline jobs is varied.

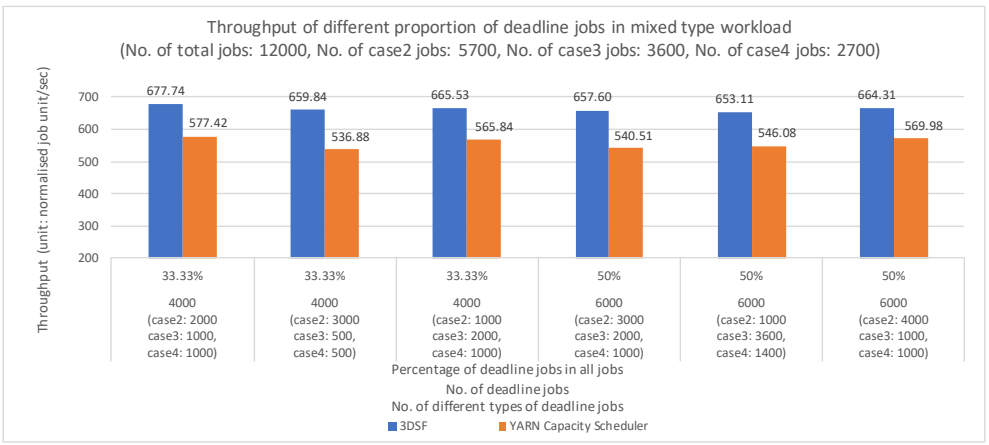

FIGURE 25 Throughput on a mixed cases workload

Figure 25 presents the throughput results of running mixed cases. Our 3DSF's average throughput is 663.02 job units/s while Capacity Scheduler's is only 556.12 job units/s. 3DSF improved upon Capacity Scheduler's throughput by 19.22%. When the percentage of total jobs for deadline jobs is 33.33%, the average throughput of 3DSF is 667.70 job units/s while Capacity

Scheduler's throughput is only 560.05 job units/s. When the fraction of deadline jobs is 50%, 3DSF's throughput reduces to 658.34 job units/s. When the proportion of deadline jobs in the total jobs is smaller, the throughput is higher.

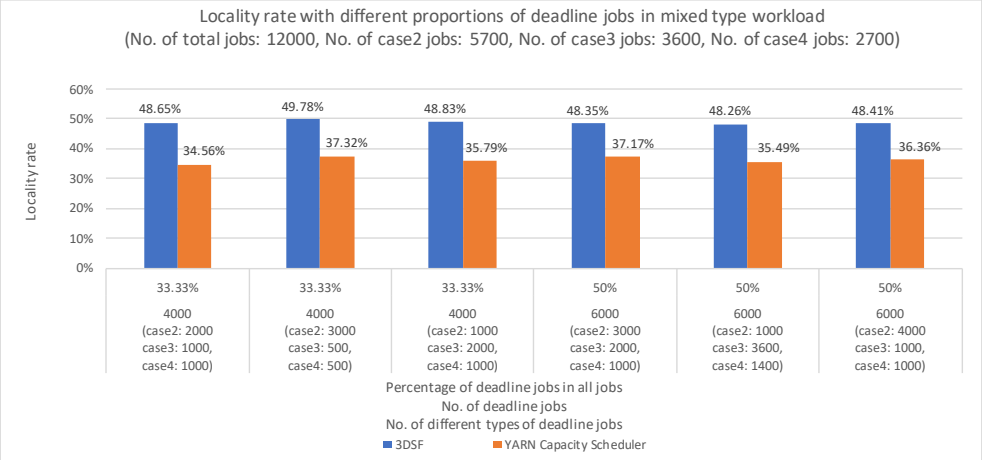

FIGURE 26 Locality rate on a mixed cases workload

Figure 26 presents the locality rate of running mixed cases. 3DSF's average locality rate is 48.71% and the YARN Capacity Scheduler's average locality rate is 36.12%, with 3DSF improving on the YARN scheduler's locality rate by 12.60%.

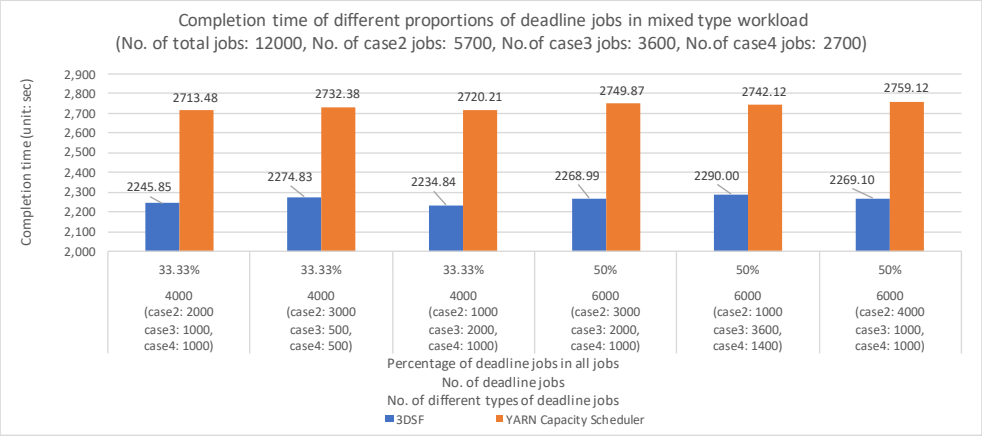

FIGURE 27 Completion time on a mixed cases workload

Figure 27 shows the completion times of different job combinations. The average completion time of 3DSF is 2,263s and the average completion time of Capacity Scheduler is 2,736s. 3DSF can reduce completion time by 17.26%, compared with Capacity Scheduler.

Figure 28 presents the resource utilisation of running mixed cases. This workload is of a mixed type, and thus all 3 kinds of resource utilisation are relatively high. Using 3DSF, the average CPU utilisation is 90.72%, the average memory utilisation is 88.74% and the average I/O utilisation is 84.42%. Compared with our 3DSF, Capacity Scheduler's average CPU utilisation is 80.41% and is lower by 10.31%, Capacity Scheduler's average memory utilisation is 77.09% and is lower by 11.65%, and Capacity Scheduler's average disk I/O utilisation is 71.43% and is lower by 12.99%.

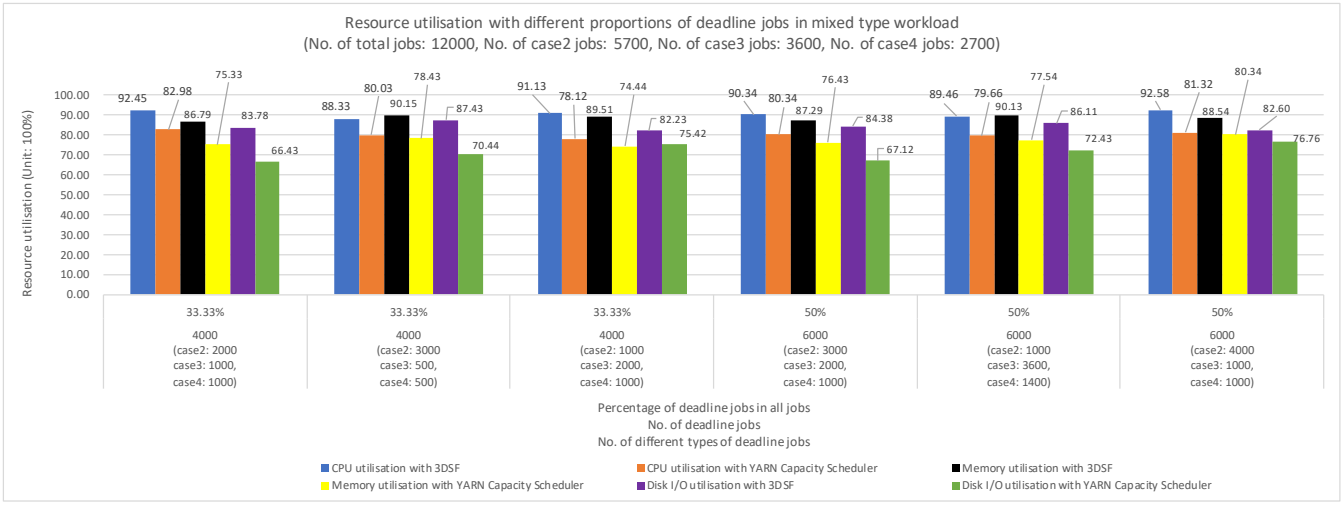

FIGURE 28 Resource utilisation on a mixed cases workload

10 | THREATS TO VALIDITY

Construct validity threats include the wrong choice of evaluation metrics, incorrect collection of metrics, and the incorrect inference of performance from these metrics. We chose well-known and accepted metrics in cloud computing system performance and scheduling evaluation. We collected data using YARN provided APIs and scheduler statistics, and ran jobs using the different schedulers with the same experimental set-up, data collection and data analysis. We only used a cloud cluster which had 5 nodes to employ experiments. Real experimental environments could indicate that experimental results is more reliable, in order to prove our scheduling framework's superiority. Furthermore, scaling out a cloud cluster will provide more resources to the jobs, which enables more jobs to be running in parallel. According to the features of our scheduling framework, scaling will improve our framework's throughput and performance as well as accelerate scheduling capability. But scaling could be infinite. Therefore, we believe our scale is sufficient.

Internal validity threats include our choice of job mixes, interaction of different resource needs in mixed-jobs, and our cloud platform resources available. We tried to mitigate these threats by using a variety of job mixes, a representative cloud platform and configuration, and running different sets of experiments with different workload, resource, job type and job number mixes.

The key **external** validity threat to our experiments of its generalisablility of the results are due to the limited number of benchmarks and real-world applications that we have run them on. We chose a mix of benchmarks, and a mixture of quite different real-world applications, and a mixture of job types to mitigate this.

11 | CONCLUSIONS AND FUTURE WROK

The dynamic nature of cluster environments and the varying computing workloads affect the execution time and computational resources used in the scheduling process. A better resource allocation approach ensures that all tenants receive system resources in a fair manner, which improves overall utilisation and throughput and reduces traffic in an over-crowded system. Scheduling optimisation is an important approach to improving this data locality to reduce the tradeoff and completion time. Moreover, for real-time applications and services, deadline is also a major criterion in judging the QoS. From the perspective of tenants, meeting the deadline requirement can improve a system's QoS and accelerate a system's reputation. The challenge for scheduling strategies is that the fairness of resource allocation often collides with data locality and deadline-based QoS. To realise these objectives, our novel work proposes a deadline constrained and data locality aware dynamic scheduling framework, named ***3DSF***. Our ***3DSF*** is a generic framework, without platform limitations. Our scheduling framework considers users' deadline-based QoS requirements, cloud system's performance and resource allocation to improve resource utilisation, system's throughput, to reduce the completion time of jobs and to better meet their QoS. Our ***3DSF*** contains (1) a real-time, preemptive, deadline constrained job scheduler, (2) an optimised data locality aware scheduler and (3) an improved Dominant

Resource Fairness (DRF) greedy resource allocation approach with 3-dimensional demand vector <CPU, memory, disk I/O> and (4) an integration of the above-mentioned schedulers together and an auto-adaptive suite. The suite automatically chooses a corresponding scheduling strategy or a combination of scheduling strategies to satisfy the profiles, resource requirements, data locality requirements and specific QoS requirements of different jobs in heterogeneous clouds. From the resource management perspective, a suite of novel scheduling algorithms is proposed to deal with different types of workloads in multi-tenancy clouds. These algorithms can maximise cloud resource usage, improve system's throughput and also minimise the makespan of jobs
(the overall completion time of workload). Our experimental results indicate that ***3DSF*** can improve upon the deadline-based QoS by approximately 16%, can improve the throughput by approximately 22%, the locality rate by approximately 13%, and the resource utilisation by at least 11%, while reducing the completion time by approximately 15%.

In future, our ***3DSF*** will consider the privacy and security issue and the relevance of data of multi-tenants. In addition, we will also extend our dynamic resource allocation strategy of multi-dimensional resources to tasks, based on analysing the requirements of jobs and real-time system's resource utilisation for maximisation.

ACKNOWLEDGMENT

This research is supported by a scholarship from Swinburne University of Technology.

References

1. Vavilapalli VK, Murthy A, Douglas C, et al. Apache Hadoop Yarn: Yet another resource negotiator. In: 4th ACM annual Symposium on Cloud Computing. ; 2013: 5–16.

2. Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: 8th USENIX Conference on Networked Systems Design and Implementation. ; 2011: 22–35.

3. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; 51(1): 107–113.

4. Zaharia M. *An architecture for fast and general data processing on large clusters*. Morgan & Claypool . 2016.

5. Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A. Quincy: fair scheduling for distributed computing clusters. In: 22nd ACM SIGOPS symposium on Operating systems principles. ; 2009: 261–276.

6. Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J. Omega: flexible, scalable schedulers for large compute clusters. In: 8th ACM European Conference on Computer Systems. ; 2013: 351–364.

7. White T. *Hadoop: The definitive guide*. O'Reilly Media . 2012.

8. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review* 2007; 41(3): 59–72.

9. Jia R, Grundy J, Keung J. Software engineering for multi-tenancy computing challenges and implications. In: ACM International Workshop on Innovative Software Development Methodologies and Practices. ; 2014: 1–10.

10. Kumara I, Han J, Colman A, Heuvel v. dWJ, Tamburri DA, Kapuruge M. SDSN@RT: A middleware environment for single-instance multitenant cloud applications. *Software: Practice and Experience* 2019; 49(5): 813-839.

11. Ellens W, Akkerboom J, Litjens R, Van Den Berg H. Performance of cloud computing centers with multiple priority classes. In: 5th IEEE International Conference on Cloud Computing (CLOUD). ; 2012: 245–252.

12. Remesh Babu KR, Samuel P. Service-level agreement–aware scheduling and load balancing of tasks in cloud. *Software: Practice and Experience* 2019.

13. Jain N, Menache I, Naor JS, Yaniv J. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. *ACM Transactions on Parallel Computing* 2015; 2(1): 3:1–3:29.

14. Zhu J, Gao B, Wang Z, et al. A dynamic resource allocation algorithm for database-as-a-service. In: 2011 IEEE International Conference on Web Services. ; 2011: 564–571.

15. Gohad A, Ponnalagu K, Narendra NC. Model driven provisioning in multi-tenant clouds. In: 2012 IEEE Annual SRII Global Conference. ; 2012: 11–20.

16. paper wA. An Overview of the AWS Cloud Adoption Framework. In: https://d1.awsstatic.com/whitepapers/aws_cloud_adoption_framework.pdf. AWS; 2017.

17. Wang X, Devadhar V, Murugesan P. Providing a routing framework for facilitating dynamic workload scheduling and routing of message queues for fair management of resources for application servers in an on-demand services environment. 2016. US Patent 9,348,648.

18. Qu H, Mashayekhi O, Terei D, Levis P. Canary: A Scheduling Architecture for High Performance Cloud Computing. *arXiv preprint arXiv:1602.01412* 2016.

19. Singh S, Chana I. QRSF: QoS-aware resource scheduling framework in cloud computing. *The Journal of Supercomputing* 2015; 71(1): 241–292.

20. Babaioff M, Mansour Y, Nisan N, et al. ERA: A framework for economic resource allocation for the cloud. In: 26th ACM International Conference on World Wide Web Companion. ; 2017: 635–642.

21. Liu N, Li Z, Xu J, et al. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In: 37th IEEE International Conference on Distributed Computing Systems (ICDCS). ; 2017: 372–382.

22. Rimal BP, Maier M. Workflow scheduling in multi-tenant cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems* 2017; 28(1): 290–304.

23. Kao YC, Chen YS. Data-locality-aware MapReduce real-time scheduling framework. *Journal of Systems and Software* 2016; 112: 65–77.

24. Jia R, Grundy J, Yang Y, Keung J, Li H. Providing Fairer Resource Allocation for Multi-tenant Cloud-Based Systems. In: 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom). ; 2015: 306–313.

25. Benifa , Bibal J. Performance improvement of mapreduce for heterogeneous clusters based on efficient locality and replica aware scheduling (ELRAS) strategy. *Wireless Personal Communications* 2017; 95(3): 2709–2733.

26. Singh AK, Dziurzanski P, Mendis HR, Indrusiak LS. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Computing Surveys (CSUR)* 2017; 50(2): 24:1–24:40.

27. Peng Y, Bao Y, Chen Y, Wu C, Guo C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: 13th EuroSys Conference. ; 2018: 3–16.

28. Sandholm T, Lai K. Dynamic proportional share scheduling in Hadoop. *Job Scheduling Strategies for Parallel Processing* 2010: 110–131.

29. He S, Guo L, Guo Y, Wu C, Ghanem M, Han R. Elastic application container: A lightweight approach for cloud resource provisioning. In: 26th IEEE International Conference on Advanced information networking and applications (AINA). ; 2012: 15–22.

30. Song Y, Sun Y, Shi W. A two-tiered on-demand resource allocation mechanism for VM-based data centers. *IEEE Transactions on Services Computing* 2013; 6(1): 116–129.

31. Ghodsi A, Zaharia M, Shenker S, Stoica I. Choosy: Max-min fair sharing for datacenter jobs with constraints. In: 8th ACM European Conference on Computer Systems. ; 2013: 365–378.

32. Hadoop Fair Scheduler. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

33. Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

34. Grandl R, Chowdhury M, Akella A, Ananthanarayanan G. Altruistic scheduling in multi-resource clusters. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). ; 2016: 65–80.

35. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In: 8th USENIX Conference on Networked Systems Design and Implementation. ; 2011: 323–336.

36. Chen H, Lin W, Kuo Y. MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems. *IEEE Transactions on Cloud Computing* 2018; 6(1): 127–140.

37. Sahni J, Vidyarthi P. A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment. *IEEE Transactions on Cloud Computing* 2018; 6(1): 2–18.

38. Duggan M, Shaw R, Duggan J, Howley E, Barrett E. A multitime-steps-ahead prediction approach for scheduling live migration in cloud data centers. *Software: Practice and Experience* 2019; 49(4): 617-639.

39. Cheng C, Li J, Wang Y. An energy-saving task scheduling strategy based on vacation queuing theory in cloud computing. *Tsinghua Science and Technology* 2015; 20(1): 28–39.

40. Xiong K, Perros H. Service performance and analysis in cloud computing. In: IEEE World Conference on Services-I. ; 2009: 693–700.

41. Khabbaz M, Assi C. Modelling and Analysis of A Novel Deadline-Aware Scheduling Scheme for Cloud Computing Data Centers. *IEEE Transactions on Cloud Computing* 2018; 6(1): 141–155.

42. Abhaya VG, Tari Z, Zeephongsekul P, Zomaya A. Performance analysis of EDF scheduling in a multi-priority preemptive M/G/1 queue. *IEEE Transactions on Parallel and Distributed Systems* 2014; 25(8): 2149–2158.

43. Cheng D, Zhou X, Xu Y, Liu L, Jiang C. Deadline-aware mapreduce job scheduling with dynamic resource availability. *IEEE Transactions on Parallel and Distributed Systems* 2018; 30(4): 814–826.

44. Cheng D, Rao J, Jiang C, Zhou X. Resource and deadline-aware job scheduling in dynamic hadoop clusters. In: 29th IEEE International Parallel and Distributed Processing Symposium. IEEE. ; 2015: 956–965.

45. Chen W, Rao J, Zhou X. Preemptive, low latency datacenter scheduling via lightweight virtualization. In: USENIX Annual Technical Conference (USENIX ATC 17). ; 2017: 251–263.

46. Liu L, Zhou Y, Liu M, et al. Preemptive Hadoop jobs scheduling under a deadline. In: 8th IEEE International Conference on Semantics, Knowledge and Grids (SKG). ; 2012: 72–79.

47. Chung A, Park JW, Ganger GR. Stratus: cost-aware container scheduling in the public cloud. In: ACM Symposium on Cloud Computing. ; 2018: 121–134.

48. Zheng Y, Ji B, Shroff N, Sinha P. Forget the deadline: Scheduling interactive applications in data centers. In: 8th IEEE International Conference on Cloud Computing (CLOUD). ; 2015: 293–300.

49. Delgado P, Didona D, Dinu F, Zwaenepoel W. Kairos: Preemptive data center scheduling without runtime estimates. In: 9th ACM Symposium on Cloud Computing. ; 2018.

50. Venkataraman S, Panda A, Ananthanarayanan G, Franklin MJ, Stoica I. The power of choice in data-aware cluster scheduling. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). ; 2014: 301–316.

51. Agrawal K, Li J, Lu K, Moseley B. Scheduling parallel DAG jobs online to minimize average flow time. In: 27th annual ACM-SIAM symposium on Discrete algorithms. ; 2016: 176–189.

52. Özkaya MY, Benoit A, Uçar B, Herrmann J, Catalyurek U. A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning. 2019.

53. Grandl R, Kandula S, Rao S, Akella A, Kulkarni J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). ; 2016: 81–97.

54. Jin J, Luo J, Song A, Dong F, Xiong R. Bar: An efficient data locality driven task scheduling algorithm for cloud computing. In: 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). ; 2011: 295–304.

55. Xue R, Gao S, Ao L, Guan Z. BOLAS: Bipartite-Graph Oriented Locality-Aware Scheduling for MapReduce Tasks. In: 14th IEEE International Symposium on Parallel and Distributed Computing (ISPDC). ; 2015: 37–45.

56. Lim N, Majumdar S, Ashwood-Smith P. MRCP-RM: a technique for resource allocation and scheduling of MapReduce jobs with deadlines. *IEEE Transactions on Parallel and Distributed Systems* 2017; 28(5): 1375–1389.

57. Li C, Zhang J, Ma T, Tang H, Zhang L, Luo Y. Data locality optimization based on data migration and hotspots prediction in geo-distributed cloud environment. *Knowledge-Based Systems* 2019; 165: 321–334.

58. Kaur K, Kumar N, Garg S, Rodrigues JJ. EnLoc: data locality-aware energy-efficient scheduling scheme for cloud data centers. In: IEEE International Conference on Communications (ICC). ; 2018: 1–6.

59. Tang Z, Zhou J, Li K, Li R. A MapReduce task scheduling algorithm for deadline constraints. *Cluster computing* 2013; 16(4): 651–662.

60. Yang Y, Xu J, Wang F, Ma Z, Wang J, Li L. A MapReduce Task Scheduling Algorithm for Deadline-Constraint in Homogeneous Environment. In: 2nd IEEE International Conference on Advanced Cloud and Big Data (CBD). ; 2014: 208–212.

61. Xiao W, Bhardwaj R, Ramjee R, et al. Gandiva: Introspective cluster scheduling for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). ; 2018: 595–610.

62. Boutin E, Ekanayake J, Lin W, et al. Apollo: scalable and coordinated scheduling for cloud-scale computing. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). ; 2014: 285–300.

63. Li T, Tang J, Xu J. A predictive scheduling framework for fast and distributed stream data processing. In: IEEE International Conference on Big Data (Big Data). ; 2015: 333–338.

64. Jia R, Yang Y, Grundy J, Keung J, Li H. A highly efficient data locality aware task scheduler for cloud-based systems. In: Proceedings of International Conference on Cloud Computing (CLOUD). ; 2019: 496–498.

65. Jia R, Yang Y, Grundy J, Keung J, Li H. A Deadline Constrained Preemptive Scheduler Using Queuing Systems for Multi-tenancy Clouds. In: International Conference on Cloud Computing (CLOUD). ; 2019: 63–67.

66. Hortonworks Data Platform: system administration guides. In: ; 2014.

67. Ousterhout K, Rasti R, Ratnasamy S, Shenker S, Chun BG, ICSI V. Making sense of performance in data analytics frameworks. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI). ; 2015: 293–307.

68. Kadirvel S, Fortes J. Grey-box approach for performance prediction in Map-Reduce based platforms. In: 21st IEEE International Conference on Computer Communications and Networks (ICCCN). ; 2012: 1–9.

69. Keras. https://keras.io/.

70. Ousterhout K, Wendell P, Zaharia M, Stoica I. Sparrow: distributed, low latency scheduling. In: 24th ACM Symposium on Operating Systems Principles. ; 2013: 69-84.

71. Chen CH, Lin JW, Kuo SY. MapReduce Scheduling for Deadline-Constrained Jobs in Heterogeneous Cloud Computing Systems. *IEEE Transactions on Cloud Computing* 2015; 6(1): 127-140.

72. Munkres J. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics* 1957; 5(1): 32–38.

73. Gao S, Xue R. BOLAS+: Scalable lightweight locality-aware scheduling for Hadoop. In: IEEE Trustcom/BigDataSE/ISPA. ; 2016: 1077–1084.

74. Wang C, Wu Q, Tan Y, Wang W, Wu Q. Locality based data partitioning in MapReduce. In: 16th IEEE International Conference on Computational Science and Engineering (CSE). ; 2013: 1310–1317.

75. Sztrik J. Basic queueing theory. *University of Debrecen, Faculty of Informatics* 2012; 193.

76. Xu X, Yu H, Pei X. A novel resource scheduling approach in container based clouds. In: IEEE 17th International Conference on Computational Science and Engineering. ; 2014: 257–264.

77. Anderson TW, Mathématicien EU. *An introduction to multivariate statistical analysis*. 2. New York: Wiley . 1958.

78. Hadoop Capacity Scheduler. *https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html*.

APPENDIX

A NOTATION LIST

In this work, we define some notations. Table A1 lists the notations of this work. The order of the notations is based on

TABLE A1 Notation index in this work

Notation	Definition	Notation	Definition
Res	total resources capacities	Com	consumed resources
Dos_m	tenant m's dominant shares	All_m	tenant m's total allocated resources
Dem_m	the demand of the next task that tenant m wants to launch	G	weighted bipartite graph
T	the set of tasks	S	the set of cluster nodes
E	the set of edges between tasks and nodes	wei_i	the set of edges' weights
u^i_{res}	resource utilisation of Node i	$u^i_{ava\_res}$	available resource utilisation of Node i
edge $e(t, s)$	task $t \in T$ is placed on node $s \in S$	$S^G_{pre}(t)$	the set of task t's preferred nodes in G
$f\ T \rightarrow S$	Allocate task t to node $f(t)$	α	the allocation for task t
$\sum_{t \in T} \alpha(t)$	current total allocation	$u^{\alpha(t)}_{res}$	node $\alpha(t)$'s resource utilisation
cap_t	task t's transmission capacity	$rtt_t(n_i, n_j)$	the round-trip delay time for TCP connection between node n_i and node n_j
$T^{tran}_t(d_t)$	data transfer time	d_t	the data block (split) which task t receives
$f(x)$	ReLu activation function	$C(t, \alpha(t))$	the cost of task t which is processed on node $\alpha(t)$
C_{loc}	the local cost	C_{rem}	the remote cost
T^{com}_{total}	the total completion time of jobs	$W^{rel}_{\alpha(t)}$	the resource releasing time
$Dist(n_s, n_d)$	the number of hops between nodes n_s and n_d	$W^{soj-rem}$	the remaining execution time for running tasks
W^{soj}	estimated execution time of tasks	$W^{soj-alr}$	the already running time of tasks
λ_i	arrival rate	Dem_i	required resource of job_i
t^{arr}_i	arrival time of job_i	t^{dea}_i	the deadline of job_i

p_i	regular job's priority	μ_k	service rate
L_{sys}	total number of jobs in cloud, including the ones waiting in the queue and being serviced. $\bar{L}_{sys}$ is the mean of L_{sys}	L_{wai}	total number of jobs waiting in the queue, excluding the ones being serviced. $\bar{L}_{wai}$ is the mean of L_{wai}
L_{ser}	total number of jobs being serviced. $\bar{L}_{ser}$ is the mean of L_{ser}	L	average number of jobs in the queue at any time
W_i^{wai}	the time that job_i waits in the queue, excluding the time that job_i spends on service. $\bar{W}_{wai}$ is the mean waiting time	W_i^{soj}	job_i's sojourn time, including the time that job_i waits in the queue and are in service. $\bar{W}_{soj}$ is the mean sojourn time
T_{ser}	average service time of a job, denoted as $\frac{1}{\mu}$	$\sigma^2(\bar{L}_{wai})$	variance of mean number of jobs waiting in the queue, excluding the ones being serviced