2019 IEEE International Conference on Cloud Computing, June 8-13, Milan, Italy (c) IEEE 2019

A Deadline Constrained Preemptive Scheduler Using Queuing Systems for Multi-tenancy Clouds

Jia Ru*, Yun Yang*, John Grundy†, Jacky Keung‡ and Li Hao§
*School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia 3122
Email:{rjia, yyang}@swin.edu.au
†Faculty of Information Technology, Monash University, Melbourne, Australia 3145
Email:john.grundy@monash.edu
‡Department of Computer Science, City University of Hong Kong, Hong Kong SAR
Email:Jacky.Keung@cityu.edu.hk
§Amaris.AI Pte.Ltd, Singapore, 089760
Email:liucoolhao@gmail.com

***Abstract*—Scheduling on clouds is required so that service providers can meet Quality of Service (QoS) requirements of tenants. Deadline is a major criterion in judging QoS. This work presents a real-time, preemptive, constrained scheduler using queuing theory – PDSonQueue – which enables better meetinhg of QoS requirements. PDSonQueue also shortens a job's completion time and improves system's throughput. PDSonQueue, as a dynamic priority real-time greedy scheduler, builds a queuing-based mathematical model to accurately predict a job's execution and waiting time, where jobs arrive by following a stochastic process and request resources. Our scheduler introduces a novel "Earliest Maximal Waiting Time First (EMWTF)" concept to fine tune job scheduling to guarantee the job being accomplished within the deadline. 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. PDSonQueue integrates an improved Dominant Resource Fairness (DRF) greedy resource allocation approach to capture the essence of tenants' resource allocation and run as many jobs as possible. Our experimental results indicate that PDSonQueue can improve by at least 20% of deadline-based QoS rate, and by at least 30% for throughput.**

***Index Terms*—deadline, multi-tenancy, scheduling, queuing theory, resource preemption**

I. Introduction

Cloud computing provides a variety of QoS sensitive services to different tenants in a scalable and virtualised manner [1]. E.g., financial applications such as stock trades have strict job deadlines and cloud service consumers are willing to pay for this. In contrast, many scientific jobs are willing to trade a bounded delay for lower costs. The primary challenge is to meet the Service Level Agreements (SLAs) for their job completion deadline [2]. Scheduling mechanisms should differentiate jobs based on their importance or priority [2].

A better scheduling policy should not only enable cloud resources to be better provisioned according to tenants' needs, but also minimise the number of violated SLAs. It also enables the cloud system to maximise resource usage and achieve high throughput and provides end users quicker response. Current research has paid much more attention to the multi-tenancy scheduler, such as Fair Scheduler [3], where basic features such as data locality, user priority, fault-tolerance and fairness are all considered. To date only a few algorithms handle such a deadline constraint [2], [4]. Although work in [5] first formulates the preemptive scheduling problem under deadline constraint, its job execution time estimator is very simple and does not consider run time complexity and input data size. Preemption is an important technology to avoid delaying production jobs while allowing the system to be shared by other non-production jobs [5]. However, existing schedulers seldom consider preemption, so jobs have to wait for completion of others, which may result missing their deadlines. Accurately predicting user's service performance avoids over provision to meet SLAs.

This study focuses on scheduling and resource allocation for deadline constrained jobs. We have deadline as our primary objective to enable more deadline constrained jobs to have their QoS met. We try to minimise the number of jobs that miss their deadlines. This work also improves deadline-based QoS and system's throughput with no degradation of SLAs. Our key contributions in this work are: 1) a precise job service time and waiting time estimator; 2) a deadline constrained preemptive scheduler that takes users' deadlines as part of input and sorts the deadline constrained jobs based on Earliest Maximal Waiting Time First (EMWTF); 3) use of a preemption mechanism to enable deadline constrained jobs to preempt resource from regular jobs to guarantee deadline jobs being completed before the deadlines; and 4) integration of a fairer Dominant Resource Fairness (DRF) resource allocation strategy to maximise the number of jobs being allocated.

In this paper, Section II discusses the related work. Section III presents our preemptive deadline constrained scheduler, PDSonQueue. Section IV presents experimental results. Section V concludes the work and outlines future research.

II. Related work

The predictability of jobs' service time helps deadline constrained jobs to be finished on time. When we know a job's likely execution time in advance, it helps us to determine when to run deadline constrained jobs in order 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 [2], [4]. Work in [6] constructs performance models by running the jobs on different numbers of cluster nodes and fitting a regression model to the observed performance. Such performance models still make many assumptions that each job is run multiple times by users of the cluster, so that the cost of experiment-driven model building can be amortised over a large number of runs. Work in [5] proposes a 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. However, the job execution cost model only considers the number of available slots and input data sizes.

Earliest Deadline First (EDF) is a dynamic priority real-time scheduling algorithm that considers time constraints of tasks in scheduling for execution [7], [8]. Work in [7] uses a queuing theoretic performance model for a multi-priority preemptive M/G/1/EDF system. Their proposed model predicts the mean waiting time for a given class based on 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. To our best knowledge, the preemption performance in current preemptive schedulers such as [5], [7] is limited and most cloud deadline schedulers rarely consider preemption.

Queuing models are widely used to model service performance in clouds, aiming to optimise energy consumption, resource allocation, performance prediction, resource management, load sharing, etc. [9]. Work in [10] 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 for a cloud resource provider when different clients negotiate different SLAs. Work in [9] adopts vacation M/G/1 queuing system with exhaustive service to model the task schedule and analyse the expectations of task's sojourn time and energy consumption of nodes under steady state. However, none is currently used in deadline scheduling.

III. Our Proposed PDSonQueue

Although EDF algorithms are popular in guaranteeing job deadlines in real-time systems, they are not effective in dynamic cloud environments. For instance, a long job has a longer deadline than that of short job, but the long job's execution time is also longer than a short job. According to job's maximal waiting time (the difference of deadline and execution time), the long job may have less time to wait for running. Thus, even if with a longer deadline, the long job still needs to be executed earlier than the short job. Within maximal waiting time, a job gets necessary resources to run, which guarantees this job will be finished before its deadline. As 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 are put ahead of long job to be run. This may result in a long job failing to get resources during the waiting time and it cannot be finished on time. If using EMWTF, a long job that has less waiting time can be put ahead to be run before shorter jobs. Short jobs then have more waiting time to gain resources, so they also may be finished before their deadline. We use an M/M/n mathematic queuing model to model job service to estimate service time, waiting time, assuming an exponential density function for the inter-arrival and service times. We predict a job's maximal waiting time before it is executed.

A. Overview of PDSonQueue

Preemptive scheduling heuristics judiciously accept, schedule and suspend real-time services to maximise a system's QoS performance. Complimenting previous non-preemptive algorithms, 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.

PDSonQueue includes a greedy resource allocation strategy, a preemption mechanism to preempt resources for deadline constrained jobs when available resources are not sufficient, and a queuing model to estimate jobs' service time and waiting time, as shown in Fig. 1. A tenant has a variety of users (1). Users submit jobs to the cloud cluster (2). Jobs specify their key resource requirements: <CPU, memory, vdisk> and QoS - *deadline*. Jobs are classified as *deadline jobs* and *regular jobs*. Regular jobs do not have deadline demand and are divided into low and high priority jobs. Correspondingly, 2 queues are built in our scheduler: deadline queue (3) sorts deadline jobs based on EMWTF and regular queue (4) sorts regular jobs based on First Come First Serve (FCFS). According to different dominant resource consumption, our scheduler divides each of the queues into three sub-queues (5): CPU-, memory- and I/O-intensive sub-queues. E.g., a CPU-intensive deadline job should be put into the CPU-intensive deadline sub-queue. The deadline jobs with earliest waiting time should be run first. When a job applies for resources successfully, a container is created with a timestamp (6). A container is a logical bundle of resources bound to a particular cluster node [1]. If deadline jobs cannot get enough resources during their waiting time, they preempt resources from regular, low priority jobs (7). Low priority jobs will be suspended, the corresponding containers will be suspended and relevant resources will be released until deadline jobs get enough resources (8). If all low priority jobs are suspended but there are still insufficient resources, high priority jobs will be preempted (9). Our preemptive strategy chooses the latest served jobs to preempt and suspends the containers based on timestamps and dominant share resource profile. Latest created containers with the same dominant share as a deadline job will be suspended first. Our resource allocation uses our previous work (10) – 3-dimensional demand vector <CPU, memory, vdisk> greedy DRF algorithm [1] to enhance fairer resource sharing and optimise the number of

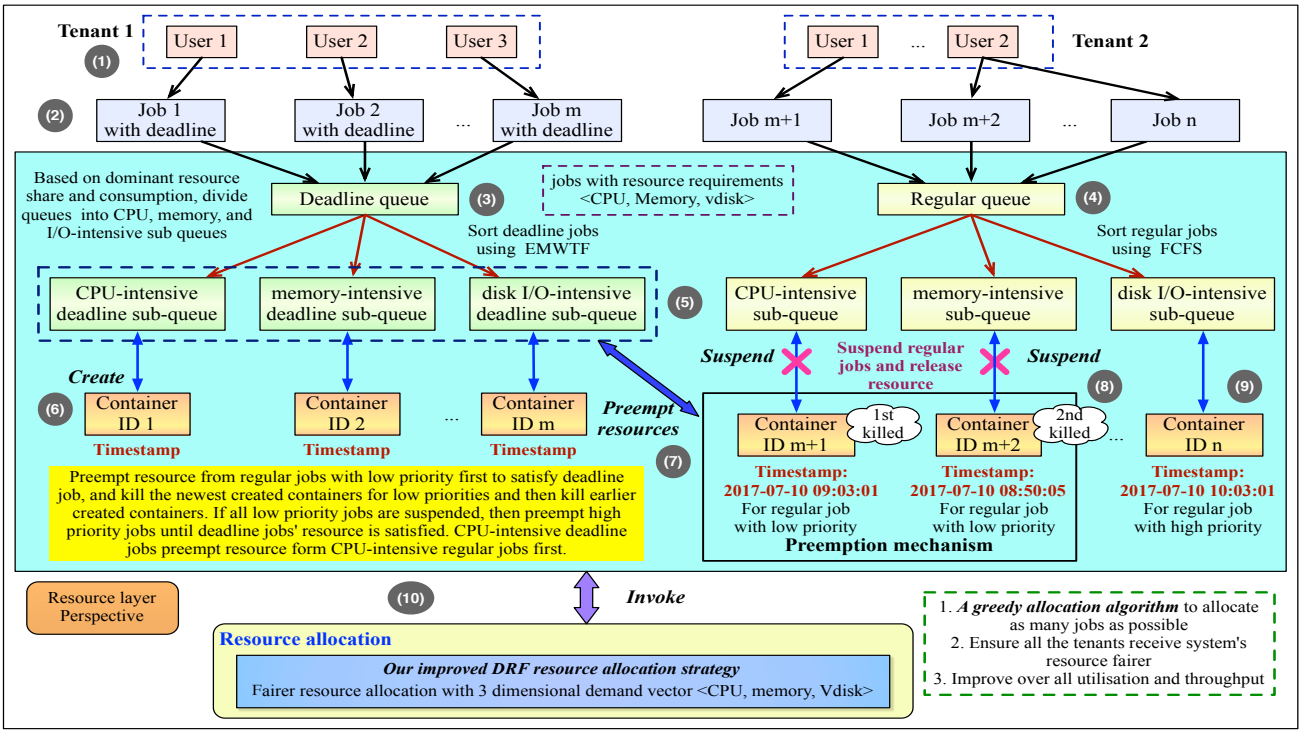

Fig. 1. Deadline constrained scheduling model based on M/M/n queue model

jobs being serviced in each local sub-queue, which maximises the number of jobs in a global cloud queuing system.

B. Mathematical analysis of job scheduling

A cloud has n heterogeneous nodes, donated as N_1, N_2,..., N_n. Submitted jobs are served independently by n nodes [11]. As arriving jobs come from different tenants and users, the inter-arrival time can be modelled as an exponential random variable. The job arrival rate and service rate on nodes can be obtained through long-term statistical results or a number of experiments. We assume the arrival of the jobs conforms to a Poisson process λ_i. The service rate is also assumed to be independent and exponential with parameter μ_k [11], where:

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

The mean service rate of node j is μ, and mean service rate of entire cloud system is $n\mu$. When $\frac{\lambda}{n\mu} < 1$, the theory [11] has proven the cloud system being stable, marking $\rho_1 = \frac{\lambda}{\mu}$, $\rho = \frac{\lambda}{n\mu}$. The service is the same as job arrival rate that follows Poisson process.

A multi-tenant cloud provides end users with different functionality but with potentially distinct QoS values, so jobs' characteristics and requirements are varied and heterogeneous [1]. Jobs are classified as CPU-, memory-, and I/O intensive based on resource requirements and consumption; jobs are preemptive, which means execution on any node can be suspended, except deadline jobs; and the service time of jobs are not known to the scheduler a prior. 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.

1) Steady state equation: A cloud system should provide its services continually and will not restrict the number of the jobs. When the state of the cloud system is k $(0 \le k \le n)$, 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. When the system is stable, let $\rho = \frac{\lambda}{n\mu}$ and stability condition $\rho < 1$. Stationary probability p_k for state k can be determined by solving a set of balance equations, which state the flux into a state should be equal to the flux out of this state when the system is stationary [10]:

$$p_k = \begin{cases} \frac{\rho_1^k}{k!} p_0 = \frac{n^k}{k!}\rho^k p_0, & 0 \le k < n \\ \frac{\rho_1^k}{n! n^{k-n}} p_0 = \frac{n^n}{n!}\rho^k p_0; & k \ge n \end{cases} \tag{1}$$

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

2) Mean queue length, sojourn time and waiting time: The mean number of jobs that are being serviced is below:

$$\begin{aligned} \bar{L}_{ser} &= \bar{k} = \sum_{k=0}^{n} k p_k + n \sum_{k=n+1}^{\infty} p_k \\ &= \sum_{k=0}^{n} k \bullet \frac{n^k}{k!}\rho^k p_0 + \sum_{k=n+1}^{\infty} n \bullet \frac{n^n}{n!}\rho^k p_0 \\ &= n\rho(\sum_{k=0}^{n-1} p_k + \sum_{k=n}^{\infty} p_k) = n\rho = \rho_1 \end{aligned} \tag{2}$$

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

$$\begin{aligned} \bar{L}_{wai} &= \sum_{k=n}^{\infty} (k-n) p_k = \sum_{h=0}^{\infty} h p_{h+n} = \frac{\rho(n\rho)^n}{n!} p_0 \sum_{h=1}^{\infty} h\rho^{h-1} \\ &= \frac{\rho\rho_1^n}{n!(1-\rho)^2} p_0 = \frac{\rho_1^{n+1}}{(n-1)!(n-\rho_1)^2} p_0 \end{aligned} \tag{3}$$

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

$$\bar{L}_{sys} = \bar{L}_{wai} + \bar{L}_{ser} = \bar{L}_{wai} + \rho_1 = \frac{\rho\rho_1^n p_0}{n!(1-\rho)^2} + \rho_1 \tag{4}$$

When a queuing system reaches statistical equilibrium and $L = \lambda \bar{W}_{soj}$, $\bar{L}_{wai} = \lambda \bar{W}_{wai}$, it obeys Little's Law [11]. Thus, we get:

$$\begin{aligned} \bar{W}_{wai} &= \frac{\bar{L}_{wai}}{\lambda} = \frac{\rho_1^n p_0}{\mu n \bullet n!(1-\rho)^2} \\ \bar{W}_{soj} &= \frac{\bar{L}_{sys}}{\lambda} = \bar{W}_{wai} + T_{ser} = \bar{W}_{wai} + \frac{1}{\mu} \end{aligned} \tag{5}$$

The probability of that a job must wait in the queue to gain service from the cloud, $P(Waiting)$, is referred as Erlang's C formula [11]:

$$P(Waiting) = \sum_{k=n}^{\infty} p_k = \sum_{k=n}^{\infty} \frac{n^n}{n!} \rho^k p_0 = C(n, \rho) \tag{6}$$

$$C(n, \rho_1) = \sum_{k=n}^{\infty} p_n \bullet \rho^{k-n} = \frac{p_n}{1-\rho} = \frac{n p_n}{n - \rho_1} \tag{7}$$

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

$$E(L_{ser}) = \frac{\lambda}{\mu} = \rho \tag{8}$$

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

$$\begin{aligned} E(L_{wai}) = & E(L_{wai} | q \geq n) P(q \geq n) \\ & + E(L_{wai} | q < n) P(q < n) \end{aligned} \tag{9}$$

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

$$E(L_{wai}) = C(n, \rho) \frac{\rho}{n - \rho} \tag{10}$$

An arriving job has to wait if at its arrival the number of jobs in the system is at least n and the time while a customer is serviced is exponentially distributed with parameter $n\mu$. 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 [11]

$$f_w(x) = \sum_{j=0}^{\infty} p_{n+j} (n\mu)^{j+1} \frac{x^j}{j!} e^{-n\mu x} \tag{11}$$

If the arriving number of 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:

$$f_s(x) = P(No\ waiting) \mu e^{-\mu x} + f_{w+ser}(x) \tag{12}$$

C. Schedular Algorithm

When new job_m comes, we estimate job_m's waiting time W_m^{wai} and sojourn time W_m^{soj} (*Line:4*). If job_m is deadline constrained, it is put in the deadline queue (*Lines:6-7*). Deadline job_i which has the smallest W_i^{wai} are executed first (*Line:11*). 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:9*). Within its W_i^{wai}, job_m needs to successfully apply for the required resource Dem_i from the cloud system. If job_m is a normal job, it is put in a regular queue based FCFS. When system's available resource Res_{rem} is smaller than job_i's required resource Dem_i (*Line:18*) and W_i^{wai} is smaller than resource released time (the remaining service time of next job being finished) (*Line:19*), the system preempts 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 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 Dem_i, we suspend 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 job_i is a deadline job, within W_i^{wai} at each iteration, the scheduler selects the job with the lowest dominant share ready to run (*Line:43*). During each iteration, the user's task with lowest dominant share Dem_i is ready to run (*Line:44*). If job_i as a deadline job nearly exhausts its waiting time W_i^{wai}, deadline job_i which has the highest priority gets Dem_i resource and runs immediately, without regarding to the issue which user's job has smallest dominant share and should be run next. If all the low priority jobs from $queue_{reg}^{inten[*]}$ are suspended, but updated Res_{rem} is still smaller than Dem_i, the latest low priority jobs being run from other sub queues will be suspended to satisfy Dem_i. If all the low priority jobs 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} is larger than Dem_i, the system allocates Dem_i amount resource to job_i (*Line:21*). The iterations of resource preemption and allocation must be finished in W_i^{wai}. When Res_{rem} is smaller than Dem_i and W_i^{wai} is smaller than resource released time, if Dem_i is larger than next to be finished job's (job_{nexfin}'s) resource demand, Dem_{nexfin}, plus Res_{rem}, preemption starts (*Lines:23-25*). If Dem_i is smaller than Dem_{nexfin} plus Res_e, job_i waits until $job_{nextfin}$ is finished (*Lines:26-29*).

Algorithm 1 PDSonQueue Scheduler

1: **// Scheduling phase**
2: **while** (jobs $\notin \phi$) **do**
3: record job_i=$\{Dem_i, t_i^{arr}, t_i^{dea}\}$ or job_i=$\{Dem_i, t_i^{arr}, p_i\}$
4: estimate the 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 FCFS 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 $queue_{reg}^{inten[*]}$ until Dem_i resource is available
21: allocate required resource Dem_i to job_i
22: **else**
23: **if** ($Dem_i > (Res_{rem} + Dem_{nexfin})$) **then**
24: suspend the newest running low priority regular jobs job_{newlo} in $queue_{reg}^{inten[*]}$ until Dem_i resource is available
25: allocate required resource Dem_i to job_i
26: **else**
27: job_i waits until next job being finished job_{nexfin} is finished and job_{nexfin}'s resource is released
28: allocate required resource Dem_i to job_i
29: **end if**
30: **end if**
31: **else** /***Dem_i is smaller than Res_{rem}***/
32: no preemption; allocate required resource Dem_i to job_i
33: **end if**
34: **else** /***job_i is a regular job***/
35: **if** ($Dem_i > Res_{rem}$) **then**
36: job_i waits until other resource released; after releasing enough resource, allocate required resource Dem_i to job_i
37: **else**
38: allocate required resource Dem_i to job_i

39: **end if**
40: **end if**
41: **// Resource allocation phase**
42: Call our previous work - improved DRF allocation strategy [1]
43: **while** ($W_i^{wai} > 0$) **do**
44: select user z with lowest dominant share Dos_z
45: **end while**
46: preempt resource and allocate required resource Dem_i to job_i
47: **return** job_i begins to run

IV. Experiments

We compare PDSonQueue's performance to YARN Fair Scheduler [3], using 3 metrics: deadline-based QoS, throughput and completion rate. To test *throughput* metric, we use "job unit" to generalise jobs and the normalised job unit is 10ms. The deadline setting is followed by normal distribution. Our cloud cluster contains 5 machines each with 16GB of RAM, 2.9 GHz 8 cores Intel Processors, 3 1TB disks, running Hadoop 2.6.0. We select 3 benchmarks: 1) **Pi estimator** is a pure ***CPU-intensive*** application that employs a Monte Carlo method to estimate the value of Pi. 2) **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. 3) **Read/Write file** is a simple ***I/O-intensive*** task that reads and writes files repeatedly.

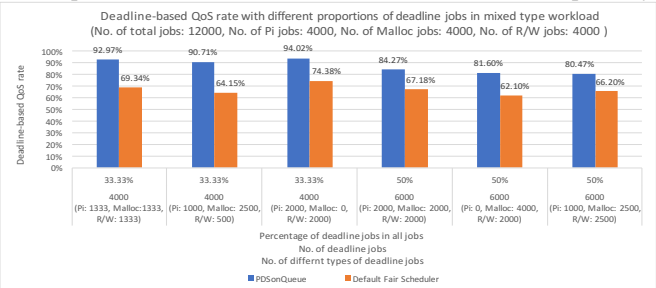

Fig. 2. Deadline-based QoS rate on a mixed workload

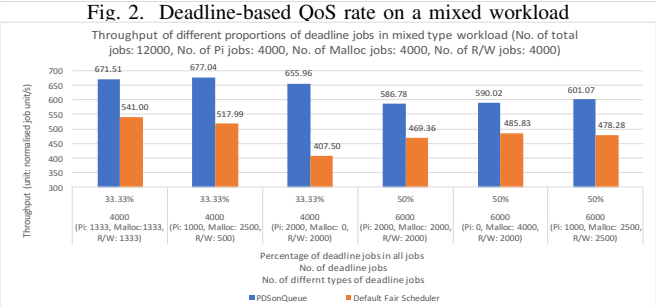

Fig. 3. Throughput on a mixed workload

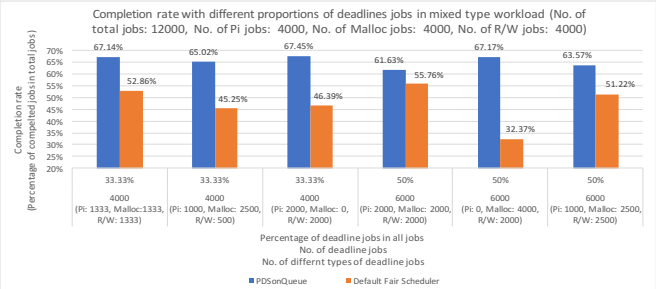

Fig. 4. Completion rate on a mixed workload

We set deadline jobs occupying 33% and 50% and changing the number of deadline Pi estimation jobs, Malloc jobs and read/write jobs. Fig. 2 shows the QoS achievement. When the proportion of deadline jobs is 33%, the average QoS of PDSonQueue is 92.57% regardless of the fraction of different types of deadline jobs and that of fair scheduler is 69.29%. When deadline jobs occupy 50% of all jobs, our scheduler's average QoS rate is 82.11% and fair scheduler's QoS rate is 65.16%. With the increase of deadline jobs, our scheduler's QoS is decreased by 10% but average QoS achieved is still very good at 87.34%. However, fair scheduler's QoS fluctuates and averages around 65%, which is lower by 22.34% than PDSonQueue, and our algorithm's performance is steady. In Fig. 3, PDSonQueue's average throughput is higher (630.39 job units/s) than that of fair scheduler (483.32 job units/s), with 30.43% improvement. When deadline jobs are less (33%), our algorithm's average throughput is even higher (668.16 job units/s). In Fig. 4, PDSonQueue's average completion rate is 65.33% and fair scheduler's is 47.31%. Our algorithm has 18.02% improvement, which shows PDSonQueue can complete more jobs in time. PDSonQueue's performance is much better than fair scheduler.

V. Conclusions and Future Work

Our novel work proposes a real-time preemptive deadline constrained scheduler, *PDSonQueue*, which enables more jobs to satisfy their deadline-based QoS requirements; allocates more jobs to be processed; and improves system's throughput. We use queuing model to accurately estimate jobs' execution time and available waiting time. PDSonQueue uses Earliest Maximal Waiting Time First to sort constrained jobs to guarantee deadline jobs to be executed by their deadlines. It uses a preemption mechanism to avoid the delay of high priority constrained jobs while allowing the system's resource to be shared by regular jobs. Lastly, PDSonQueue integrates an improved DRF resource allocation approach to run as many jobs as possible. Our experimental results show that PDSonQueue performs much better. In future, we will extend our dynamic resource allocation scheduler of multi-dimensional resources to tasks.

VI. Acknowledgement

This work is supported in part by Swinburne University of Technology, Monash University, the General Research Fund of the Research Grants Council of Hong Kong (No.11208017) and the research funds of City University of Hong Kong (9678149, 7005028, and 9678149), and the Research Support Fund by Intel (9220097).

References

[1] R. Jia, J. Grundy, Y. Yang, J. Keung, and H. Li, "Providing fairer resource allocation for multi-tenant cloud-based systems," in *2015 IEEE 7th Int. Conf. Cloud Computing Technology and Science,*, pp. 306–313.
[2] H. Chen, W. Lin, and Y. Kuo, "MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems," *IEEE Trans on Cloud Computing*, vol. 6, no. 1, pp. 127–140, 2018.
[3] "Hadoop fair scheduler," in *https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html*.
[4] J. Sahni and P. Vidyarthi, "A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment," *IEEE Trans. on Cloud Computing*, vol. 6, no. 1, pp. 2–18, 2018.
[5] L. Liu, Y. Zhou, M. Liu, G. Xu, X. Chen, D. Fan, and Q. Wang, "Preemptive Hadoop jobs scheduling under a deadline," in *2012 8th Int. Conf. Semantics, Knowledge and Grids,*. IEEE, 2012, pp. 72–79.
[6] A. Aboulnaga, Z. Wang, and Z. Y. Zhang, "Packing the most onto your cloud," in *2009 ACM 1st Int. WS Cloud Data Management,*. ACM, 2009, pp. 25–28.
[7] V. G. Abhaya, Z. Tari, P. Zeephongsekul, and A. Y. Zomaya, "Performance analysis of edf scheduling in a multi-priority preemptive m/g/1 queue," *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2149–2158, 2014.
[8] M. Khabbaz and C. Assi, "Modelling and analysis of a novel deadline-aware scheduling scheme for cloud computing data centers," *IEEE Trans. on Cloud Computing*, vol. 6, no. 1, pp. 141–155, 2018.
[9] C. Cheng, J. Li, and Y. Wang, "An energy-saving task scheduling strategy based on vacation queuing theory in cloud computing," *Tsinghua Science and Technology*, vol. 20, no. 1, pp. 28–39, 2015.
[10] W. Ellens, J. Akkerboom, R. Litjens, H. van den Berg *et al.*, "Performance of cloud computing centers with multiple priority classes," in *2012 IEEE 5th Int. Conf. Cloud Computing*. IEEE, 2012, pp. 245–252.
[11] J. Sztrik, "Basic queueing theory," *University of Debrecen, Faculty of Informatics*, vol. 193, 2012.