Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

A resource management technique for processing deadline-constrained multi-stage workflows

Norman Lim[1*], Shikharesh Majumdar[1] and Peter Ashwood-Smith[2]

Abstract

The use of cloud computing that provides resources on demand to various types of users, including enterprises as well as engineering and scientific institutions, is growing rapidly. An effective resource management middleware is necessary to harness the power of the underlying distributed hardware in a cloud. Two of the key operations provided by a resource manager are resource allocation (matchmaking) and scheduling. This paper concerns the problem of matchmaking and scheduling an open stream of *multi-stage jobs* (or *workflows*) with Service Level Agreements (SLAs) on a cloud or cluster. Multi-stage jobs require service from multiple system resources and are characterized by multiple phases of execution. This paper presents a resource allocation and scheduling technique called RM-DCWF: Resource Management Technique for Deadline-constrained Workflows that can efficiently matchmake and schedule an open stream of multi-stage jobs with SLAs, where each SLA is characterized by an earliest start time, an execution time, and a deadline. A rigorous simulation-based performance evaluation of RM-DCWF is conducted using synthetic workloads derived from real scientific workflows. In addition, the impact of various system and workload parameters on system performance is investigated. The results of this performance evaluation demonstrate the effectiveness of RM-DCWF as captured in a low number of jobs missing their deadlines.

Keywords: Resource allocation and scheduling on clouds, Multi-stage jobs with SLAs, Workflows with SLAs, Jobs with deadlines

Introduction

Over the past few years, distributed computing paradigms such as cluster computing and cloud computing have been generating a lot of interest among consumers and service providers as well as researchers and system builders. For example, a number of reputable financial institutions and market research organizations have predicted a multi-billion-dollar market for the cloud computing industry [1, 2]. An important feature of cloud computing is that it allows users to acquire resources on demand and pay only for the time the resources are used. Investigating and devising effective *resource management techniques* for clouds and clusters is necessary to harness the power of the underlying distributed hardware and to achieve the performance objectives of a system [3], which can include generating high job throughput and low job response times, meeting the quality of service (QoS) requirements of jobs that are often captured by a service level agreement (SLA), and maintaining a high resource utilization to generate adequate revenue for the cloud service provider. As in the case of grids, a predecessor of cloud computing that also supported resources on demand, QoS and satisfying SLAs remain an important issue for cloud computing [3, 4]. Handling of jobs with a SLA often leads to an advance reservation request [5] that is characterized by an *earliest start time*, a required *execution time*, and a *deadline* for completion. This is of critical importance for latency-sensitive business and scientific applications that can include live business intelligence and sensor-based applications which rely on a timely processing of the collected data. Two of the key operations that a resource manager needs to provide are *resource allocation* (*matchmaking*) and *scheduling*. Note that the matchmaking and scheduling operations are jointly referred to as

* Correspondence: nlim@sce.carleton.ca
[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada
Full list of author information is available at the end of the article

mapping operation [6]. Given a pool of resources, the matchmaking algorithm chooses the resource(s) to be allocated to an incoming job. Once a number of jobs are allocated to a specific resource, a scheduling algorithm determines the order in which jobs allocated to the resource should be executed for achieving the desired system objectives. Performing effective matchmaking and scheduling is difficult because the SLA of the jobs need to be satisfied, while also considering system objectives, which can include minimizing the number of jobs that miss their deadlines as well as generating adequate revenue for the service provider.

Due to cloud computing becoming more prevalent, a variety of applications are being run on clouds, including those that are characterized by multiple phases of execution and require processing from multiple system resources (referred to as *multi-stage jobs*). Scientific applications and workflows that are used in various fields of study, such as physics and biology, are examples of multi-stage jobs that are run on clouds. Moreover, another example of a popular multi-stage application that is typically run on clouds is MapReduce [7], which is a programming model (proposed by Google) for simplifying the processing of very large and complex data sets in a parallel manner. MapReduce is used by many companies and institutions, typically in conjunction with cloud or cluster computing, for facilitating *Big Data analytics* [8–10]. The focus of this paper is to devise an efficient matchmaking and scheduling technique for processing an *open stream* of *multi-stage jobs* (*workflows*) with SLAs on a distributed computing environment with a fixed number of resources (e.g. a private cluster or a set of resources acquired a priori from a public cloud). Most existing research focuses on meeting the SLA for jobs that require processing from only a single resource or handling of a fixed number of multi-stage jobs executing on the system. There is comparatively less work available in the literature focusing on resource management for a workload comprising and open stream of multi-stage jobs with SLAs. Handling of an open stream of multi-stage jobs increases the complexity of the resource allocation and scheduling problem due to a continuous stream of jobs arriving on the system. Thus, the novel matchmaking and scheduling techniques described in this paper are expected to make a significant contribution to the state of the art.

This paper presents a novel resource allocation and scheduling technique, referred to as RM-DCWF: Resource Management Technique for Deadline-constrained Workflows, that can effectively perform matchmaking and scheduling for an open stream of multi-stage jobs with SLAs, where each SLA comprises an earliest start time, an execution time, and an end-to-end deadline. RM-DCWF decomposes the end-to-end deadline of a job into components (i.e. sub-deadlines), each of which is associated with a task in the job. The individual tasks of the job are then mapped on to the resources where the objective is to satisfy the job's SLA and minimize the number of jobs that miss their deadlines. In our preliminary work [11], a resource allocation and scheduling technique for processing deadline-constrained MapReduce [7] jobs (comprising of only two phases of execution) is described. The algorithm described in [11] can only handle MapReduce jobs. The algorithm presented in this paper is new as it focuses on a more complex resource management problem that considers jobs and workflows characterized by multiple (two or more) phases of execution as present in scientific workflows, for example. Furthermore, the jobs handled by the algorithm introduced in this paper can have various structures characterized by different precedence relationships among the respective constituent tasks that are not considered in [11]. To the best of our knowledge, none of the existing work focuses on all aspects of the resource management problem that this paper focuses on: devising a resource allocation and scheduling algorithm for multi-stage jobs with SLAs on a system subjected to an open stream of job arrivals. The main contributions of this paper include:

- A novel resource allocation and scheduling technique, RM-DCWF, for handling an open stream of multi-stage jobs with SLAs. Two task scheduling policies are devised.
- Two algorithms devised to decompose the end-to-end deadline of a multi-stage job to assign each task of the job a sub-deadline.
- Insights gained into system behavior and performance from a rigorous simulation-based performance evaluation of RM-DCWF using a variety of system and workload parameters. The synthetic workloads used in the experiments are based on real scientific workflows described in the literature.
- A comparison of the performance of the proposed technique with that of a conventional first-in-first-out (FIFO) scheduler as well as a technique from the literature [29] that has objectives similar to RM-DCWF is presented.

The rest of the paper is organized as follows. Section "Related work" summarizes related work, and Section "Problem description and resource management model" provides a description of how the resource allocation and scheduling problem is modelled. The algorithms devised to decompose the end-to-end deadline for multi-stage jobs are described in Section "Deadline budgeting algorithm for workflows". In Section "RM-DCWF's matchmaking and scheduling algorithm", RM-DCWF and its matchmaking and scheduling algorithms are presented. The results of the performance evaluation of RM-DCWF and the insights gained into system behavior are discussed in Section "Performance Evaluation of RM-DCWF". Lastly, Section "Conclusions and Future Work" presents the conclusions and offers directions for future work.

Related work

The use of distributed computing environments for processing multi-stage jobs (workflows) has received significant attention from researchers in the past few years. A representative set of existing work related to resource management on distributed systems for processing multi-stage jobs, including workflows (see Section "Resource management on distributed systems for processing workflows") and MapReduce jobs (see Section "Resource management techniques on distributed systems for processing MapReduce jobs"), is presented next.

Resource management on distributed systems for processing workflows

A representative set of existing work related to resource management on clouds for processing workflows is presented next. A workflow is usually modelled using a directed acyclic graph (DAG) where each node in the graph represents a task in the workflow and the edges of the graph represent the precedence relationships among the tasks.

The focus of [12, 13] is on describing workflow scheduling algorithms for grids. More specifically, in [12], the authors propose a workflow scheduling algorithm using a Markov decision process based approach that aims to optimally schedule the tasks of the workflows such that the number of workflows that miss their deadlines is minimized. The authors of [13] present a heuristic-based workflow scheduling algorithm, called Partial Critical Paths (PCP), whose objective is to generate a schedule that satisfies a workflow's deadline, while minimizing the financial cost of executing the workflow on a service-oriented grid.

The following papers present various techniques to schedule workflows on a cloud environment. In [14], a heuristic scheduling algorithm for clouds to process workflows where users can specify QoS requirements, such as a deadline or financial budget constraint, is presented. The objective is to ensure that the workflow meets its deadline, while the financial budget constraint is not violated. The technique described in [15] uses a particle swarm optimization (PSO) methodology to develop a heuristic-based scheduling algorithm to minimize the total financial cost of executing a workflow in the cloud. PSO is a stochastic optimization technique that is frequently used in computational intelligence. The authors of [16] also present a PSO-based technique (set-based PSO) for scheduling workflows with QoS requirements, including deadlines, on clouds. In [17], a Cat Swarm Optimization (CSO)-based workflow scheduling algorithm for a cloud computing environment is presented. The proposed algorithm considers both data transmission cost and execution cost of the workflow, and its objective is to minimize the total cost for executing the workflow.

The research described in [18] devises a resource management technique for workflows with deadlines executing on hybrid clouds. Initially, the algorithm attempts to only use the resources of the private cloud to execute the workflow. However, if the deadline of the workflow cannot be met, the algorithm decides the type and number of resources to allocate from a public cloud so as to satisfy the deadline of the workflow. The framework presented in [19] focuses on the virtual machine (VM) provisioning problem. It uses an extensible cost model and heuristic algorithms to determine the number of VMs that should be provisioned in order to execute a workflow, while considering requirements such as the completion time of the workflow. The framework uses both single and multi-objective evolutionary algorithms to perform resource allocation and scheduling for the workflows. In [20], the authors present an evolutionary multi-objective optimization-based workflow scheduling algorithm, specifically designed for an infrastructure-as-a-service platform, that optimizes both the workflow completion time and cost of executing the workflow.

In [21], the authors present a resource allocation technique based on force-directed search for processing multi-tier Web applications where each tier provides a service to the next tier and uses services from the previous tier. The focus of [22] is on scheduling multiple workflows, each one with their own QoS requirements. The authors present a scheduling strategy that considers the overall performance of the system and not just the completion time of a single workflow. In [23], the authors describe a workflow scheduling technique for clouds that considers workflows with deadlines and the availability of cloud resources at various time intervals (time slots). The motivation is that public cloud service providers do not have unlimited resources and their resources must be shared among multiple users. Thus, the scheduling algorithm has to consider the available time slots for executing the user's requests and not assume that resources are unlimited and can be used at any time.

Resource management techniques on distributed systems for processing MapReduce jobs

This section presents a representative set of work that focuses on describing resource management techniques for platforms processing MapReduce jobs with deadlines, which have become important for latency-sensitive business or scientific applications, such as live business intelligence and real-time analysis of event logs [24].

In [24], the authors present a technique called Minimum Resource Quota Earliest Deadline First with Work-

Conserving Scheduling (MinEDF-WC) for processing MapReduce jobs characterized by deadlines. MinEDF-WC allocates the minimum number of task slots required for completing a job before its deadline and has the ability to dynamically allocate and deallocate resources (task slots) from active jobs when required.

A policy for dynamic provisioning of public cloud resources to schedule MapReduce jobs with deadlines is described in [25]. Initially, jobs are executed on a local cluster and if required, resources from a public cloud are dynamically provisioned to meet the job's deadline. The authors of [26] devise algorithms for minimizing the cost of allocating virtual machines to execute MapReduce jobs with deadlines. For example, the authors present a Deadline-aware Tasks Packing (DTP) approach where the idea is to assign the map tasks and reduce tasks of jobs to execute on existing VMs as much as possible until a job cannot meet its deadline, in which case a new VM is provisioned to execute the job.

In [27], the authors focus on the joint considerations of workload balancing and meeting deadlines for MapReduce jobs. Scheduling algorithms are proposed that are based on integer linear programming and solved with a linear programming solver using a rounding approach. A new MapReduce scheduler for processing MapReduce jobs with deadlines based on bipartite graph modelling, called the Bipartite Graph Modeling MapReduce Scheduler (BGMRS), is presented in [28]. BGMRS considers nodes with varying performance (e.g., those present in a heterogeneous cloud computing environment) and is able to obtain the optimal solution of the scheduling problem for a batch workload by transforming the problem into a well-known graph problem: minimum weighted bipartite matching.

In [29], a MapReduce Constraint Programming based Resource Management technique (MRCP-RM) is presented for processing an open stream of MapReduce jobs with SLAs, where each SLA comprises an earliest start time, an execution time, and a deadline. The objective of MRCP-RM is to minimize the number of jobs that miss their deadlines. Furthermore in [30], the authors adapt the MRCP-RM algorithm and implement it on Hadoop [31], which is a popular open-source framework that implements the MapReduce programming model. Experiments are conducted on a Hadoop cluster deployed on Amazon EC2 that demonstrate the effectiveness of the resource management technique.

Comparison with related work

The related works described in this section consider multi-stage jobs (workflows) with deadlines; however, most of the works focus on scheduling a single workflow or a fixed number of workflows (i.e. a batch workload on a closed system). To the best of our knowledge, none of the existing work focuses on all aspects of the resource management problem that this paper focuses on: matchmaking and scheduling an *open stream* of multi-stage jobs (that includes both scientific workflows and MapReduce jobs) with SLAs, where each SLA is characterized by an earliest start time, an execution time and an end-to-end deadline, on a distributed computing environment, such as a set of resources acquired a priori from a public cloud.

Problem description and resource management model

This section describes how the problem of matchmaking and scheduling an open stream of multi-stage jobs with SLAs on a distributed computing environment is modelled (see Fig. 1). Such an environment can correspond to a private cluster or a set of nodes acquired a priori from a cloud (e.g., Amazon EC2) for processing the jobs. The distributed environment is modelled as a set of resources, $R = \{r_1, r_2, \ldots, r_m\}$ where m is the number of resources in the system. Each resource r in R has a capacity (c_r), which specifies the number of tasks that resource r can execute in parallel at any point in time. Note that other researchers have modelled resources in a similar manner (see [12, 14, 16, 18], for example).

The system is subject to an open stream of multi-stage jobs. Each multi-stage job j that arrives on the system is characterized by an earliest start time (s_j) and an end-to-end deadline (d_j) by which the job j should complete executing. In addition, each job j also comprises a set of tasks, where each task t has an execution time (e_t) and can have one or more precedence relationships. The multi-stage job and the precedence relationships between its tasks can be modelled using a directed acyclic graph (DAG) (see Fig. 2, for example). The nodes (vertices) of the DAG represent the tasks of the job, and the edges of the DAG show the precedence relationships between the tasks of the job.

The example multi-stage job shown in Fig. 2 is characterized by two phases of execution where the number i in the small circle indicates the i^{th} execution phase. An *execution phase* in a multi-stage job is a collection of tasks that perform a specific function in the job. Note that the execution phase that a constituent task belongs to is specified by the user when the job is submitted to the system. In the sample job shown in Fig. 2, the first phase of execution comprises three tasks: t1, t2, and t3, and the function of these three tasks is to read and parse the input data. These three tasks do not have any *direct preceding* tasks (referred to as *parent tasks*) that need to be completed before they start executing. This implies that these tasks can start executing at the job's earliest start time specified by the user. The tasks t4 and t5, which analyze and perform computation on the parsed data, are part of the second phase of execution. Each of these tasks has a parent task t0, as well as *indirect preceding tasks* t1, t2, and t3. The tasks t4 and t5 cannot

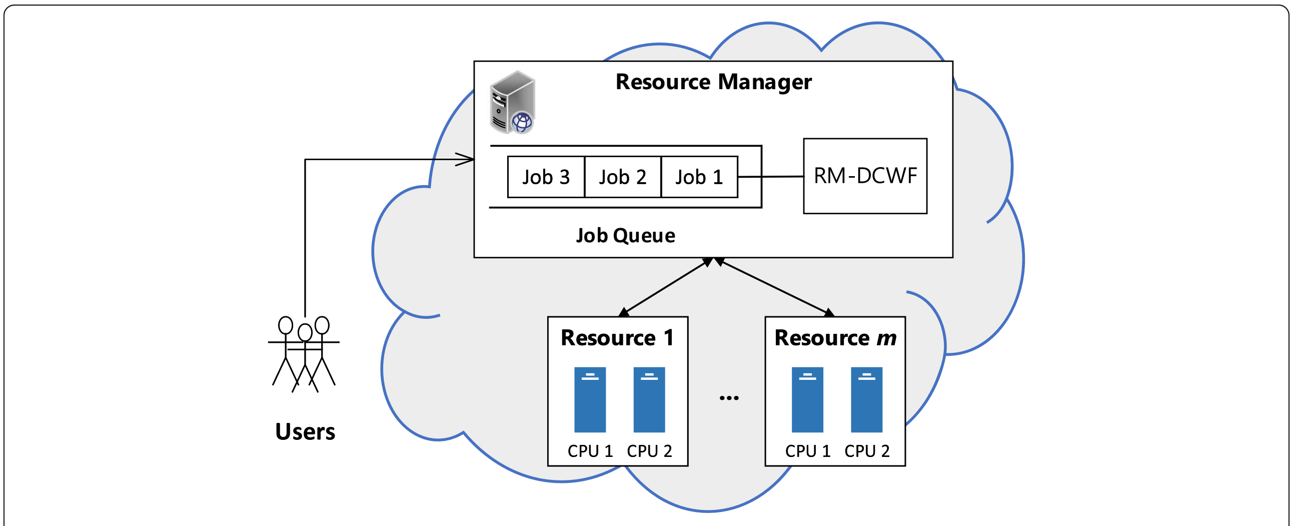

Fig. 1 Example of a system deploying RM-DCWF

start executing until task t0 finishes, which in turn cannot start executing until tasks t1, t2, and t3 finish executing. Note that some workflows are modelled using a DAG with special tasks, referred to as *dummy tasks*, whose only purpose is to enforce precedence relationships between tasks in the DAG, and thus, dummy tasks have an execution time equal to 0. For example, in Fig. 2, task t0 is a dummy task that ensures tasks in the second phase of execution start to execute only after all the tasks in the first phase have completed.

As shown in Fig. 1, jobs that arrive on the system are placed in a job queue, where jobs are sorted by non-decreasing order of their deadlines (i.e., jobs that have earlier deadlines are placed in front of jobs with later deadlines). The resource manager uses RM-DCWF, presented in this paper, to perform matchmaking and scheduling. More specifically, when the resource manager is available (i.e., not busy mapping another job) and the job queue is not empty, it removes the first job in the job queue to map onto the resources of the system, *R*. The requirements for mapping the jobs on to *R* are described

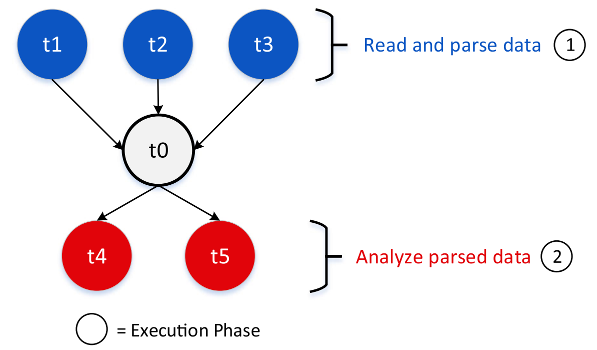

Fig. 2 DAG of a sample multi-stage job

next. The tasks of each job *j* can only execute after s_j and after their parent tasks have completed executing. In addition, each task of job *j* should complete its execution before the deadline of the job (d_j); otherwise, job *j* will miss its deadline. Note that d_j is a soft deadline, meaning that although jobs are permitted to miss their deadlines, the desired system objective is to minimize the number of late jobs. At any point in time, the number of tasks that a resource *r* in *R* can execute in parallel must be less than or equal to its capacity, c_r. A resource will execute the tasks it has been assigned in the order generated by RM-DCWF. However, a task that has been scheduled but has not started executing can be rescheduled or assigned to another resource, if required.

Deadline budgeting algorithm for workflows

Algorithm 1 presents the *Deadline Budgeting Algorithm for Workflows* (DBW algorithm), which is used by RM-DCWF to decompose the end-to-end deadline of a multi-stage job into components and to assign each task of the job a sub-deadline. The input required by the DBW algorithm is a multi-stage job *j* and two integer arguments: setOpt to indicate the approach used to calculate the *sample execution time* of the job *j* (SET_j) and laxDistOpt to specify how the *laxity* (or *slack time*) of the job *j* (L_j) is to be distributed among its constituent tasks. SET_j is an estimate of the execution time of job *j* that is calculated by the DBW algorithm and L_j is the extra time that job *j* has for meeting its deadline, if it starts executing at its earliest start time. L_j is calculated as follows:

$$L_j = d_j - s_j - SET_j \qquad (1)$$

The first step of the DBW algorithm is to calculate SET_j (line 1). SET_j is calculated using the user-estimated task execution times of the job and can be calculated in one of two ways, depending on the supplied `setOpt` argument. The first approach (`setOpt = 1`) is to calculate the execution time of job j when it executes at its maximum degree of parallelism on the set of resources R with m resources, while not considering any contention for resources (denoted SET_j^R). Recall from the previous section, the definition of R, which is a set of resources that models the distributed system job j will execute on. The approach used by the algorithm for matchmaking and scheduling the tasks of job j onto the m resources and computing SET_j^R is briefly described. The tasks of job j are allocated in non-increasing order of their execution times: the *ready* task with the highest required task execution time is allocated first; the task with the next highest execution time is considered next and so on. Tasks are considered *ready* when all of their parent tasks, as captured in the precedence graph that characterizes the workflow, have completed executing. A best-fit technique is used for allocating the tasks of the job to the resources. Each task of the job is allocated on that resource that can execute the task at its earliest possible time. Thus, the algorithm attempts to complete each task and the job at its earliest possible finish time. The second approach (`setOpt = 2`) is to calculate the execution time of the job when it executes on R, while considering the current *processing load* of the resources (i.e., considering the other jobs already executing or scheduled on R) (denoted $SET_j^{R\_PL}$). This is accomplished by scheduling the job on the system's resources to retrieve its expected completion time and then removing the job from the system. Next, the algorithm calculates the laxity of the job (L_j) using Eq. 1 (line 2). Note that when L_j is calculated using SET_j equal to SET_j^R, the laxity of the job is referred to as the *sample laxity* (SL) because the job execution time is calculated on R without considering the current processing load of the resources in R. When L_j is calculated using SET_j equal to $SET_j^{R\_PL}$, the laxity of the job is referred to as the *true laxity* (TL) because the job execution time is calculated on R while considering the current processing load of the resources in R. The final steps of the algorithm are to distribute the laxity of the job to each of its constituent tasks and to calculate a sub-deadline for each of the tasks (line 3) by invoking one of two algorithms devised: (1) the *Proportional Distribution of Job Laxity (PD) Algorithm*, which is described in Section "Proportional Distribution of Job Laxity Algorithm", and (2) the *Even Distribution of Job Laxity (ED) Algorithm*, which is discussed in Section "Even Distribution of Job Laxity Algorithm". The algorithm that is used depends on the supplied laxDistOpt input argument.

Algorithm 1: Deadline Budgeting Algorithm for Workflows
Input: job j, integer *setOpt*, integer *laxDistOpt*
Output: none

```
1: Depending on setOpt, calculate the sample execution time of job j ($SET_j$).
2: jobLaxity ← $d_j - s_j - SET_j$
3: According to laxDistOpt, invoke the PD algorithm or the ED algorithm.
```

Proportional distribution of job laxity algorithm

The PD algorithm (shown in Algorithm 2) distributes the laxity of the job to its constituent tasks according to the length of the task's execution time. This means that a task with a longer execution time is assigned a larger portion of the job's laxity, resulting in the task having a higher sub-deadline. The input required by the algorithm includes a job j to process and an integer argument, `setOpt`, to indicate how SET_j is calculated. Recall from the discussion earlier that SET_j can be calculated in one of two ways: `setOpt = 1` corresponds to SET_j^R and `setOpt = 2` corresponds to $SET_j^{R\_PL}$. A walkthrough of the algorithm is provided next.

The first step of Algorithm 2 is to calculate the sample completion time of job j (denoted SCT_j) as:

$$SCT_j = s_j + SET_j$$

where s_j is the earliest start time of job j (line 1). The second and third steps involve retrieving s_j and L_j, respectively, of the supplied job j and saving them in local variables (lines 2–3). Next, the PD algorithm performs the following operations on each task t in the job j (line 4). The first operation is to calculate the *cumulative laxity* of the task t (denoted CL_t) (line 5) as:

$$CL_t = \frac{SCT_t - s_j}{SCT_j - s_j} \times L_j$$

where SCT_t is the sample completion time of task t. Note that the sample completion time of each task is determined during the calculation of SET_j (line 1 in Algorithm 1) as follows:

$$SCT_t = s_t + e_t$$

where s_t is the scheduled start time of task t (determined by the scheduling algorithm) and e_t is the execution time of task t. The cumulative laxity of a task t is the maximum laxity that task t can have (i.e. the laxity that task t has given that none of t's preceding tasks, direct or indirect, use any of their laxities). After calculating CL_t, the sub-deadline of the task t (sd_t) is then calculated (line 6) as follows:

$$sd_t = SCT_t + CL_t \tag{2}$$

The sub-deadline of the task t is then set as shown in line 7. If the task t does not have more than one parent task, the processing of task t is complete and the algorithm moves on to process the next task; otherwise, the algorithm invokes the task's setParentTasksSubDeadlines()

method (lines 8–10). The objective of this method is to set the sub-deadline of all of *t*'s parent tasks to the sub-deadline of the task among all of *t*'s parent tasks that has the highest sub-deadline. The reason for performing this operation is that a task *t* cannot start executing until all of its parent tasks finish executing, and thus, all the parent tasks of task *t* should have the same sub-deadline. Note that after adjusting the sub-deadlines of the parents of task *t*, the sub-deadlines of the grandparents of task *t* are not altered as they do not need to be adjusted. The PD algorithm ends after processing all the tasks in the job *j*.

```
Algorithm 2: Proportional Distribution of Job Laxity Algorithm
Input: job j, integer setOpt
Output: none
1:  Depending on setOpt, calculate $SCT_j$ and store the value in sct.
2:  est ← j.getEarliestStartTime()
3:  jobLaxity ← j.getLaxity()
4:  for each task t in job j do
5:      cumulativeLaxity ← [(t.getSCT() - est) / (sct– est)] * jobLaxity
6:      subdeadline ← t.getSCT() + cumulativeLaxity
7:      t.setSubDeadline(subdeadline)
8:      if t has more than one parent task then
9:          call setParentTasksSubDeadline(t)
10:     end if
11: end for
```

Even distribution of job laxity algorithm

The ED algorithm (see Algorithm 3) does not consider the length of the task's execution time and instead distributes the laxity of the job evenly among the execution phases of the job. Recall from Section "Problem Description and Resource Management Model" that an execution phase in a multi-stage job is a collection of tasks that perform a specific function in the job. The ED algorithm requires each task in a job to have an *execution phase* attribute, which is an integer (1, 2, 3, …) that indicates the phase of execution that the task belongs to. A walk-through of the ED algorithm is provided next.

```
Algorithm 3: Even Distribution of Job Laxity Algorithm
Input: job j
Output: none
1:  jobLaxity ← j.getLaxity()
2:  executionPhases ← Get the number of execution phases in job j.
3:  laxPerEP ← jobLaxity / executionPhases
4:  Create an empty map, cumulativeLaxities <execution phase, cumulative laxity>.
5:  for i = 1 to executionPhases do
6:      cl ← i * laxPerEP
7:      cumulativeLaxitites.put(i, cl)
8:  end for
9:  for each task t in job j do
10:     ep ← t.getExecutionPhase()
11:     cumulativeLaxity ← cumulativeLaxities.get(ep)
12:     subDL ← t.getSCT() + cumulativeLaxity
13:     t.setSubDeadline(subDL)
14:     if t has more than one parent task then
15:         call setParentTasksSubDeadline (t)
16:     end if
17: end for
```

The input required by the algorithm is a job *j* to process. The first step is to retrieve the laxity of the job and save the value in a local variable (line 1). Next, the algorithm retrieves the number of execution phases in job *j* and stores the value in a variable named executionPhases (line 2). This is accomplished by checking the execution phase attribute of each task *t* in job *j*. The laxity that each execution phase should be assigned is then calculated as follows:

$$L_j^{ep} = L_j / n_j^{ep}$$

where n_j^{ep} is the number of execution phases in job *j* (line 3). The cumulative laxity for each execution phase, which is the maximum amount of laxity that an execution phase can have, is then calculated as shown in lines 4–8. More specifically, the cumulative laxity of each execution phase *ph* for a job *j* is calculated as:

$$CL_j^{ph} = ph \times L_j^{ep}$$

where *ph* is an integer in the set {1, 2, 3, …, n_j^{ep}} that represents the execution phase. A map data structure named `cumulativeLaxities` is used to store the cumulative laxity for each execution phase, where the *key* is the execution phase and the *value* is the cumulative laxity. The last phase of the algorithm (lines 9–17) uses the cumulative laxity values to calculate and assign a sub-deadline for each of job *j*'s tasks. The following operations are performed on each task *t* of job *j*. First, the execution phase of the task *t* is retrieved as shown in line 10. The cumulative laxity of the task is then retrieved from the cumulativeLaxities map using the value of the execution phase as the key (line 11). Next, the sub-deadline of the task is calculated using Eq. 2 and assigned to the task (lines 12–13). Similar to the PD algorithm, the ED algorithm invokes `setParentTasksSubDeadlines()` if the task *t* has more than 1 parent task. After all the tasks of job *j* are processed the algorithm ends.

RM-DCWF's matchmaking and scheduling algorithm

This section describes RM-DCWF's matchmaking and scheduling algorithm (also referred to as the *mapping* algorithm), which is composed of two sub-algorithms: (1) the *Job Mapping algorithm* (discussed in Section "Job Mapping Algorithm") and (2) the *Job Remapping algorithm* (described in Section "Job Remapping Algorithm"). When there is a job *j* available to be mapped, the Job Mapping algorithm is invoked. If the Job Mapping algorithm is unable to schedule job *j* to meet its deadline, the Job Remapping algorithm is invoked to remap job *j* and a set of jobs that may have caused job *j* to miss its deadline.

Job mapping algorithm

The Job Mapping algorithm is comprised of two methods: (1) `mapJob()` presented in Algorithm 4 and (2) `mapJob-Helper()` described in Algorithm 5. Note that the variables shown in the algorithms that are underlined indicate that the variables are fields belonging to RM-DCWF instead of being local variables. A walkthrough of `mapJob()` is provided first, followed by a description of `mapJobHelper()`. The input required by mapJob() comprises the following: a job to map *j*, an integer `setOpt`, an integer `laxDistOpt`, and an integer tsp. Note that except for the argument tsp, which specifies the *task scheduling policy*, these are the same input arguments used by the DBW algorithm. The method returns true if the job *j* is scheduled to meet its deadline; otherwise, false is returned.

The first step of `mapJob()` is to invoke the DBW algorithm to decompose the end-to-end deadline of the job *j* and assign each of job *j*'s tasks a sub-deadline (line 1). Next, RM-DCWF's `rootJob` field is set to *j* (line 2). The `rootJob` field stores the current job that is being mapped by RM-DCWF. The third step is to clear RM-DCWF's `prevRemapAttempts` list (line 3), which stores the various sets of jobs that a job remapping attempt has previously processed. RM-DCWF's `jobComparator` field, which specifies how jobs that need to be remapped are sorted, is then set to the *Job Deadline Comparator* (line 4) to sort jobs by non-decreasing order of their respective deadlines. A more detailed discussion of the purpose of these fields, which are used by the Job Remapping algorithm, is provided in the next section. In line 5, RM-DCWF's `taskSchedulingPolicy` field, which specifies how tasks are scheduled, is initialized. Two task scheduling policies are devised. *TSP1* schedules tasks to execute at their earliest possible times, and *TSP2* schedules tasks to execute at their latest possible times such that the tasks meet their respective sub-deadlines. The last step is to invoke Algorithm 5: `mapJobHelper()` (line 6).

```
Algorithm 4: RM-DCWF algorithm's mapJob()
Input: job j, integer setOpt, integer laxDistOpt, integer tsp
Output: a Boolean: true if the job j is scheduled to meet its deadline; false, otherwise.
1:  call DBW(j, setOpt, laxDistOpt)
2:  rootJob ← j
3:  Clear the prevRemapAttempts list.
4:  Set jobComparator to the Job Deadline Comparator.
5:  Set taskSchedulingPolicy ← tsp
6:  return mapJobHelper(j, true, true)
```

A walkthrough of `mapJobHelper()`, which performs the allocation and scheduling of job *j* onto the set of resources in the system, is provided next. The input required by `mapJobHelper()` includes the following: a job *j* to map, a Boolean `isRootJob`, which is set to true if this is the first time job *j* is being mapped; otherwise, it is set to false, and a Boolean `checkDeadline`, which is set to true if the method should try to map job *j* to meet its deadline; otherwise, it is set to false and the method has to map job *j* on the system, but it does not have to schedule job *j* to meet its deadline. The `mapJobHelper()` method starts by initializing the local variable `isJobMapped` to true (line 1). Next, all of job *j*'s tasks that need to be mapped are sorted in non-increasing order of their respective execution times (line 2), where ties are broken in favour of the task with the earlier sub-deadline. If the tasks also have the same sub-deadline, the task with the smaller task id (a unique value) is placed ahead of the task with the larger id. The method then attempts to map each of job *j*'s tasks (lines 3–4) by performing the following operations for each task *t* in job *j*. First, the startTime variable is initialized by invoking the task *t*'s `getEarliestStartTime()` method (line 5), which returns the time that task *t* can start to execute while considering any precedence relationships that *t* has. If `getEarliestStartTime()` returns −1, it means that an earliest start time for task *t* cannot be determined as yet because not all of task *t*'s parent tasks have been scheduled. In this case, `mapJobHelper()` stops processing task *t* for the moment and attempts to map the next task in job *j* (line 6). On the other hand, if startTime is not −1, the processing of task *t* continues. If RM-DCWF's `taskSchedulingPolicy` field is set to TSP2 (line 7), the expected start time of the task is updated and set as shown in line 8. The expected completion time of the task is then calculated using the expected start time of the task (line 9).

After calculating the expected start time and completion time of task *t*, the method checks whether *t* has an execution time equal to 0 (i.e., checks if task *t* is a dummy task, defined in Section "Problem Description and Resource Management Model") (line 10). If task *t* is a dummy task, it does not need to be scheduled on a resource because it has an execution time equal to 0 and only the task's scheduled start time and completion time need to be set (line 11). The task *t* is also added to the RM-DCWF's `mappedTasks` list (line 12), which stores all the tasks that have been successfully mapped for job *j*. On the other hand, if task *t* has an execution time greater than 0 (line 13), the method attempts to find a resource *r* in *R* that can execute *t* at its expected start time. If *t* cannot be scheduled to execute at its expected start time, the task is scheduled at the next best time depending on the value of the `taskSchedulingPolicy` field (line 14). If `taskSchedulingPolicy` is set to TSP1, the method schedules task *t* at its next earliest possible time. On the other hand, if `taskSchedulingPolicy` is set to TSP2, the method schedules the task at its next latest possible time, while ensuring the task's sub-deadline is satisfied.

If a resource *r* cannot be found to complete executing task *t* before job *j*'s deadline, it means job *j* cannot be mapped to meet its deadline in the current iteration. Thus, if the supplied input argument `checkDeadline` is set to true (line 15), `mapJobHelper()` attempts to remap job *j* and a set of jobs that may have caused *j* to miss its deadline by performing the following operations (lines 16–18). First, the `removePartiallyMappedJob()` method is invoked to remove each of the tasks stored in the `mappedTasks` list from the system (line 16). Algorithm 7: `remapJob()`

(described in more detail in the next section) is then invoked and the return value is saved in a variable called `isJobMapped` (line 17). The next step (line 18) is then to go to line 27 to check the value of isJobMapped. If `isJobMapped` is set to true, meaning the job has been successfully scheduled to meet its deadline, the `mappedTasks` list is cleared, job *j* is added to RM-DCWF's `mappedJobs` list (line 28), and true is returned (line 29). Otherwise, isJobMapped is set to false, meaning job *j* cannot be scheduled to meet its deadline (line 30). This leads to `mapJobHelper()` being re-invoked, but this time with the `checkDeadline` argument set to false, which will map job *j* even if it misses its deadline (line 31). False is then returned (line 32) to indicate job *j* will not meet its deadline.

Algorithm 5: RM-DCWF algorithm's *mapJobHelper()*

```
Input: job j, Boolean isRootJob, Boolean checkDeadline
Output: a Boolean: true if the job j is scheduled to meet its deadline; false, otherwise.
1:  isJobMapped ← true
2:  Sort job j's tasksToMap list in non-increasing order of the execution time of the task.
3:  while the tasksToMap list is not empty do
4:      for each task t in job j's tasksToMap list do
5:          startTime ← t.getEarliestStartTime()
6:          if startTime = -1 then continue
7:          if taskSchedulingPolicy = TSP2 then
8:              startTime ← t.getSubDeadline() – t.getExecutionTime()
9:          endTime ← startTime + t.getExecTime()
10:         if startTime = endTime then
11:             t.setScheduledTime(startTime, endTime)
12:             mappedTasks.add(t)
13:         else
14:             Find a resource r in R that can execute t at its requested time or the next
                best time depending on taskSchedulingPolicy.
15:             if t cannot be mapped to meet j's deadline and checkDeadline = true then
16:                 call removePartiallyMappedJob()
17:                 isJobMapped←remapJob(job, isRootJob)
18:                 goto line 27
19:             else
20:                 Map t on r.
21:                 mappedTasks.add(t)
22:             end if
23:         end if
24:     end for
25:     tasksToMap.removeAll(mappedTasks)
26: end while
27: if isJobMapped = true then
28:     mappedTasks.clear(); mappedJobs.add(j)
29:     return true
30: else
31:     call mapJobHelper(job, true, false)
32:     return false
33: end if
```

If either of the conditions shown in line 15 are not true (i.e., a resource is found that can complete executing task *t* before job *j*'s deadline or the input argument `checkDeadline` is false), it means that task *t* can be scheduled to execute on resource *r* (line 20) and *t* is then added to the `mappedTasks` list (line 21). The next task of job *j* is then processed by repeating lines 3–26. This sequence of operations continues until all of job *j*'s tasks are mapped on the system. After all of job *j*'s tasks have been mapped, lines 27–29 are executed (as described earlier), and then the method returns.

Job remapping algorithm

The Job Remapping algorithm is comprised of two methods: (1) `remapJob()` presented in Algorithm 6 and (2) `remapJobHelper()` outlined in Algorithm 7. A discussion of `remapJob()` is provided first, followed by a discussion on `remapJobHelper()`. The input arguments required by `remapJob()` include a job *j* to remap and a Boolean `isRootJob`. The `isRootJob` argument is set to true if it is the first invocation of `remapJob()` for attempting to remap job *j* in this iteration; otherwise, `isRootJob` is set to false. If job *j* and the set of jobs that may have prevented job *j* from meeting its deadline are remapped and scheduled to meet their deadlines, the method returns true; otherwise, false is returned.

Algorithm 6: RM-DCWF algorithm's *remapJob()*

```
Input: job j, Boolean isRootJob
Output: a Boolean: true if job j and the set of jobs to remap all are scheduled to meet their deadlines; otherwise, false.
1: taskSchedulingPolicy ← TSP1
2: if calling remapJobHelper(j, isRootJob) returns true then
3:     return true
4: else
5:     if isRootJob = false then return false
6:     Change jobComparator to the Job Laxity Comparator.
7:     return remapJobHelper(j, isRootJob)
8: end if
```

The first step of `remapJob()` is to set RM-DCWF's `taskSchedulingPolicy` field to TSP1 so the tasks that are remapped are scheduled to execute at their earliest possible times (line 1). The second step is to invoke Algorithm 7: `remapJobHelper()` (line 2). Recall from line 4 of Algorithm 4 ((`mapJob()`) that the `jobComparator` field, which specifies how the jobs that need to be remapped are sorted, is initially set to the *Job Deadline Comparator*. The Job Deadline Comparator sorts jobs in non-decreasing order of their respective deadlines with ties broken in favour of the job with the smaller laxity (i.e., tighter deadline). If `remapJobHelper()` returns true, `remapJob()` also returns true (line 3). On the other hand, if `remapJobHelper()` returns false, `remapJob()` continues by checking the supplied `isRootJob` argument (line 4). If `isRootJob` is false (line 5), meaning that this invocation of `remapJob()` is not for the original attempt for mapping job *j*, the method returns false to stop this particular remapping attempt from continuing (line 5). Otherwise, the method continues and RM-DCWF's `jobComparator` field is changed to the *Job Laxity Comparator* (line 6) and `remapJobHelper()` is invoked again to check if remapping the jobs in a different order can generate a schedule in which all the jobs to remap can meet their deadlines (line 7).

The Job Laxity Comparator sorts jobs by non-decreasing order of their respective *normalized laxity* with ties going in favour of the job with an earlier deadline. If the jobs have the same deadline, the job with the earlier arrival time (which is unique for each job) is given priority. The normalized laxity of a job *j* (denoted NL_j) is calculated as follows:

$$NL_j = \frac{L_j}{SET_j} \tag{3}$$

where L_j is the laxity of job *j* and SET_j is the sample execution time of job *j* (recall Section "Deadline Budgeting Algorithm for Workflows"). The reason for using NL_j instead of L_j for sorting the jobs is because L_j is not always a good indicator of how stringent the deadline of a job is. A job can have a large laxity value, but still have a very tight deadline if the job has a high execution time. For example, given two jobs: (1) job *j1* has s_{j1} equal to 0, d_{j1} equal to 6000, and SET_{j1} equal to 5000, and (2) job *j2* has s_{j2} equal to 5500, d_{j2} equal to 6000, and SET_{j2} equal to 100. Using this information along with Eq. 1 and Eq. 3, the following values can be calculated: L_{j1} is equal to 1000, L_{j2} is equal to 400, NL_{j1} is equal to 0.2, and NL_{j2} is equal to 4. As can be observed, job *j1* has a higher laxity compared to job *j2* (i.e., $L_{j1} > L_{j2}$); however, *j1*'s normalized laxity is much smaller compared to *j2*'s normalized laxity ($NL_{j1} < NL_{j2}$), meaning job *j1* has a more stringent deadline.

A walkthrough of `remapJobHelper()` (shown in Algorithm 7) is provided next. The input arguments and output value returned by `remapJobHelper()` are the same as those described for `remapJob()`. The first step of the method is to retrieve a subset of the jobs already scheduled on the system that may have caused job *j* to miss its deadline and store these jobs in the `jobsToRemap` list (line 1). This includes all the jobs in RM-DCWF's `mappedJobs` list that execute within the interval [s_j, d_j]. Next, the supplied job *j* is added to the `jobsToRemap` list (line 2) and then the `jobsToRemap` list is sorted using RM-DCWF's `jobComparator` (line 3). Since it is possible to have multiple (nested) invocations of `remapJobHelper()`, lines 4–6 determine when an invocation of `remapJobHelper()` (referred to as a *remapping attempt*) should be rejected. More specifically, before a remapping attempt is started, the method checks if RM-DCWF's `prevRemapAttempts` list, which stores the various sets of jobs that previous invocations of `remapJobHelper()` have processed, contains the same jobs (in the same order) as the `jobsToRemap` list (line 4). If this is true, the method returns false to stop the remapping attempt (line 5). On the other hand, if the remapping attempt is allowed to continue, the `jobsToRemap` list is added to RM-DCWF's `prevRemapAttempts` list (line 7). Next, the method checks if the supplied argument `isRootJob` is true (line 8), and if so, the current state of the system is saved to a set of variables (line 9). This involves saving the scheduled tasks of each resource in the system and making a copy of RM-DCWF's `mappedJobs` list. Furthermore, the scheduled start time and assigned resource for each task currently mapped on the system is saved. The reason for saving this information is because it may be changed during the job remapping attempt, and if the remapping attempt is not successful, the original state of the system has to be restored.

The next step is to remove all the jobs in `jobsToRemap` from the system (line 11), which involves removing the jobs from RM-DCWF's `mappedJobs` list and removing each task of each job from its assigned resource's `scheduledTasks` list. This needs to be done so that the jobs in `jobsToRemap` can be remapped on the system. All the jobs in `jobsToRemap` that have already missed their deadlines are then moved to a new list called `lateJobs` (line 12) so that the jobs that have not missed their deadlines can be remapped first. The jobs in `jobsToRemap` (line 13) are then remapped in the specific order as determined by the `jobComparator` (recall line 3). This is accomplished by invoking Algorithm 5: `mapJobHelper()` as shown in line 14. If mapJobHelper() returns true, the method maps the next job in `jobsToRemap`. If at any point `mapJobHelper()` returns false (line 14), it means that one of the jobs in `jobsToRemap` cannot be scheduled to meet its deadline and the job remapping attempt has failed. The method then checks if `isRootJob` is true (line 15), and if so, the state of the system that is saved in line 9 is restored (line 16). False is then returned to indicate that the remapping attempt has failed (line 18). On the other hand, if all the jobs in `jobsToRemap` are successfully remapped to meet their deadlines, the next step is to perform mapping for the jobs in `lateJobs` (i.e., the jobs that have missed their deadlines). This is accomplished by invoking `mapJobHelper()` (Algorithm 5) with the `checkDeadline` input argument set to false for each of the jobs in `lateJobs` (line 21). Lastly, a value of true is returned by the method to indicate the remapping attempt is successful (line 22).

Algorithm 7: RM-DCWF algorithm's *remapJobHelper()*

```
Input: job j, Boolean isRootJob
Output: a Boolean: true if job j and the set of jobs to remap all are scheduled to
meet their deadlines; otherwise, false.
1:  jobsToRemap ← Get subset of mapped jobs that can cause j to miss its
                  deadline.
2:  jobsToRemap.add(j)
3:  Sort jobsToRemap using the jobComparator.
4:  if prevRemapAttempts list contains the same jobs in the same order as the
    jobsToRemap list then
5:      return false
6:  end if
7:  Add jobsToRemap to prevRemapAttempts list.
8:  if isRootJob = true then
9:      Save current state of the system.
10: end if
11: Remove jobs in jobsToRemap from the system.
12: Move jobs in jobsToRemap that have missed their deadlines to the lateJobs list.
13: for each job j1 in jobsToRemap do
14:     if calling mapJobHelper(j1, false, true) returns false then
15:         if isRootJob = true then
16:             Restore state of the system saved in line 9.
17:         end if
18:         return false
19:     end if
20: end for
21: Remap each job j2 in lateJobs by calling mapJobHelper(j2, false, false).
22: return true
```

Time complexity analysis of the RM-DCWF matchmaking and scheduling algorithm

As discussed in this section, the worst-case time complexity of the RM-DCWF matchmaking and scheduling algorithm is $O(n^2)$ where n is the number of jobs (or workflows) that arrive on the system. In the worst-case scenario, the algorithm will have to reschedule all the jobs in the system each time a new job arrives. For example, given $n = 100$, we have the following scenario. First, a job *j1* arrives on the system and is scheduled. Sometime later, a new job *j2* (with a deadline that is earlier than *j1*'s deadline) arrives before job *j1* is completed and the algorithm schedules *j2* and reschedules *j1*. Before jobs *j1* and *j2* complete, a new job *j3* with a deadline that is earlier than the deadlines of *j1* and *j2* arrives. The algorithm schedules *j3* and then reschedules *j1* and *j2*. This process continues for the remaining 97 jobs that arrive on the system. The total number of jobs that the algorithm schedules in this worst case is then equal to:

$$1+2+3+\ldots+n=\frac{n^*(n+1)}{2}=\frac{n^2+n}{2}$$

Thus, the worst-case time complexity of the algorithm is $O(n^2)$. Note that in practice, the time complexity of the algorithm will be lower because not all jobs will need to be rescheduled when a new job arrives on the system. For example, some jobs will have completed executing and other jobs will not need to be rescheduled because they do not contend with the same time slots as the newly arriving job. Moreover, if scheduling a job is deemed to take too long because of the large number of jobs that need to be rescheduled, it is possible to limit the number of jobs to reschedule. For example, this can be accomplished by modifying line 1 of Algorithm 7 to restrict the size of the `jobsToRemap` list. Such a modification will limit the number of jobs that can be added to the list, thus limiting the number of jobs that can be rescheduled at a given point in time.

Furthermore, an in-depth and rigorous empirical performance evaluation of the algorithms using various workloads and workload and system parameters is presented in Section "Performance Evaluation of RM-DCWF". The results of the performance evaluation (discussed in more detail in Section "Results of the Factor-at-a-Time Experiments") demonstrate that the algorithm leads to a reasonably small system overhead. For example, in experiments with a very high contention for resources, leading to an average resource utilization of 0.9, the average matchmaking and scheduling time of the algorithm was measured to be less than 0.05 s.

Performance evaluation of RM-DCWF

This section describes the two types of simulation experiments conducted to evaluate the performance of RM-DCWF. The first type of experiments (presented in Section "Results of the Factor-at-a-Time Experiments" and Section "Investigation of Using a Small Number of Resources") investigate the effect of various system and workload parameters on the performance of RM-DCWF. More specifically, *factor-at-a-time* experiments are conducted where one parameter is varied and the other parameters are kept at their default values. The second type of experiments (presented in Section "Comparison with a First-in-first-out (FIFO) Scheduler" and Section "Comparison with MCRP-RM") compare the performance of RM-DCWF with that of a FIFO Scheduler and the MapReduce Constraint Programming based Resource Management technique (MRCP-RM) described in [29], respectively. MRCP-RM has objectives that are similar to that of RM-DCWF: perform matchmaking and scheduling for an open stream of multi-stage jobs with SLAs, where each job's SLA is characterized by an earliest start time, an execution time, and an end-to-end deadline.

The rest of this section is organized as follows. The experimental setup and the metrics used in the performance evaluation are described in Section "Experimental Setup". Following this, a description of the system and workload parameters used in the factor-at-a-time experiments is provided in Section "System and Workload Parameters for the Factor-at-a-Time Experiments". The results of the experiments are then presented and discussed.

Experimental setup

The experiments are executed on a PC running Windows 10 (64-bit) with an Intel Core i5–4670 CPU (3.40 GHz) and 16 GB of RAM. Note that in the experiments, only the execution of the workload on the system is simulated. RM-DCWF and its associated algorithms are executed on the PC that was described at the beginning of this section. RM-DCWF is evaluated in terms of the following performance metrics in each simulation run:

- *Proportion of Late Jobs* (P) = N/n where N is the number of late jobs in a simulation run and n is the total number of jobs processed during the simulation.
- *Average Job Turnaround Time* (T). The turnaround time of a job j (tat_j) is $CT_j–s_j$ where CT_j is job j's completion time and s_j is j's earliest start time, respectively. Thus, $T = \sum_{j\in J}(tat_j)/n$.
- *Average Job Matchmaking and Scheduling Time* (O) is the average processing time required by RM-DCWF to partition a job's deadline among its tasks and matchmake and schedule a job. $O = \sum_{j\in J}(o_j)/n$ where o_j is the processing time required for mapping job j on the system.

Note that O is a value that is measured using Java's System.nanoTime() [32] method, whereas P and T are values produced as outputs of the simulation. The *O*-by-*T* ratio (denoted *O/T*) is used as an indicator of the processing overhead of RM-DCWF. This is an appropriate indication of the processing overhead because it puts the measured values of the algorithm runtimes (*O*) into context by considering the value of *O* relative to the mean job turnaround time (*T*).

System and workload parameters for the factor-at-a-time experiments

The workloads used in the factor-at-a-time experiments are based on real scientific applications (workflows) that have been described in the literature. More specifically, the three scientific applications that are used in the experiments are named *CyberShake*, *LIGO*, and *Epigenomics*. A brief discussion of each application that includes presenting the DAG of the workflow is provided next. A more detailed description of all three applications can be found in [33].

CyberShake is a seismology application that is created by the Southern California Earthquake Center to predict earthquake hazards in a region. The DAG of a sample CyberShake workflow is presented in Fig. 3. As shown, the workflow has five phases of execution. Recall from Section "Problem Description and Resource Management Model" that the number *i* in the small circle indicates the i^{th} execution phase. The first, second, and fourth execution phases each contain multiple tasks to execute whereas the third and fifth execution phase each only have one task to execute.

The Laser Interferometer Gravitational Wave Observatory (LIGO) Inspiral Analysis workflow is used to search for and analyze gravitational waveforms in data collected by large-scale interferometers. The input data is partitioned into multiple blocks so that the data can be analyzed in parallel. Fig. 4 shows the DAG of a sample LIGO workflow, which has 6 phases of execution. In this sample LIGO workflow there are two blocks of data being processed in parallel, where each block of data has multiple waveform data (i.e. TmptlBank tasks) to process.

The Epigenomics (Genome) workflow is used for automating several commonly used operations in genome sequence processing. Fig. 5 shows a DAG of a sample Genome workflow, which is characterized by one or more *lanes*, each of which starts with the execution of a fastQSplit task. If there is more than one lane in the workflow, as shown in the example in Fig. 5, there are two mapMerge stages. The first mapMerge stage is for merging the results within a particular lane (execution phase 6), and the second mapMerge stage (referred to as the global mapMerge stage) is for merging the results of all the lanes in the workflow (execution phase 7).

Table 1 outlines the system and workload parameters used in the factor-at-a-time experiments. These experiments investigate the effect of the following parameters on system performance: job arrival rate, earliest start time of jobs, job deadlines, and the number of resources. A walkthrough of Table 1 is provided next. Note that the distributions used to generate the parameters of the workload, including the job arrival rate, earliest start time of jobs, and job deadlines are adopted from [11, 29]. The first component of the table describes the workload. For a given workflow type (CyberShake, LIGO, or Genome), there are three job sizes, each of which has an equal probability of being submitted to the system: *small*, *medium*, and *large*, comprising 30 tasks, 50 tasks, and 100 tasks, respectively. The distributions used for generating the execution times of the tasks for each workload are described in [33]. The open stream of job arrivals is generated using a Poisson process. The arrival rates used in the experiments of a given workload type are different since each of the workloads is characterized by jobs with different execution times. The average execution time of a CyberShake

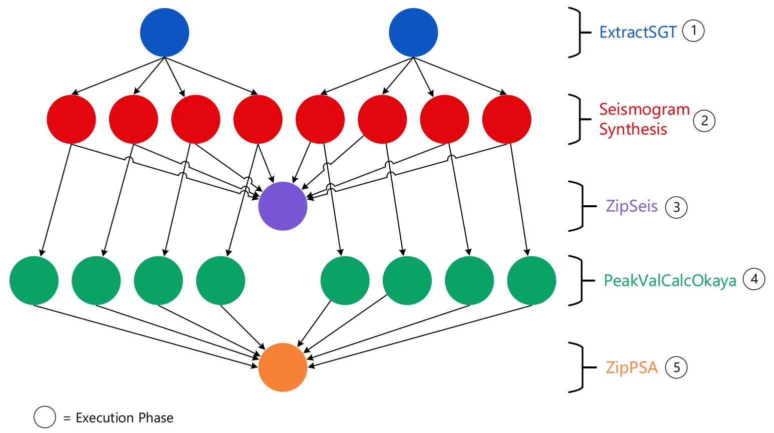

Fig. 3 DAG of a sample CyberShake application (based on [33])

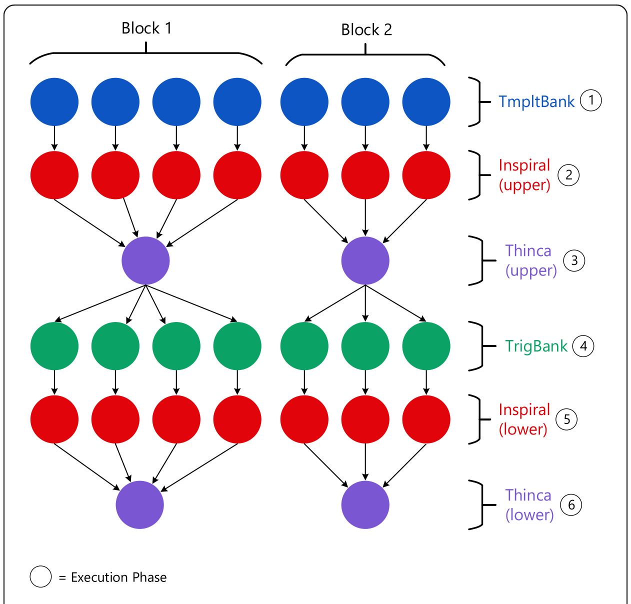

Fig. 4 DAG of a sample LIGO application (based on [33])

job, LIGO job, and Genome job on a single resource is equal to 1551 s, 13,300 s, and 160,213 s, respectively. The parameters λ_{CS}, λ_{LG}, and λ_{GN} specify the job arrival rates used for the CyberShake, LIGO, and Genome workloads, respectively. The arrival rates for each workflow are chosen such that resource utilization ranging from moderate (~50%) to moderately-high (~70%) to high (~90%) is generated on the system when using the default number of resources (50 resources, where each resource has a capacity equal to 2). The earliest start time of a job j (s_j) can be its arrival time (at_j) or at a time in the future after at_j. A random variable x, which follows a Bernoulli distribution with parameter p, is defined. The parameter p is the probability that a job j has s_j greater than at_j. If x is 0, s_j equals at_j; otherwise, s_j equals the sum of at_j and a value generated from a discrete uniform (DU) distribution with a lower-bound equal to 1 and an upper-bound equal to a parameter s_{max}. The deadlines of the jobs are generated by multiplying SET_j^R (recall Section "Deadline Budgeting Algorithm for Workflows") with an *execution time multiplier* (em) and adding the resulting value to s_j. The parameter em is used to determine the laxity (or slack time) of a job and is generated using a uniform distribution (U) where 1 is the lower-bound and em_{max} is the upper-bound of the distribution.

The remaining components of Table 1 describe the system used to execute the jobs and the configuration of RM-DCWF. The number of resources (m), which represents the number of nodes in the distributed system for processing the jobs, is varied from 40 to 50 to 60, where each resource has a capacity (c_r) equal to 2. Recall from Section "Problem Description and Resource Management Model", c_r specifies the number of tasks that a resource r can execute in parallel at any given point in time. RM-DCWF's Job Mapping algorithm can handle resources with different values of c_r as the algorithm performs resource allocation and scheduling by considering the set of all the resource slots provided by all the resources in R. Note that how the capacity of the resources, as reflected by their number of resource slots, is distributed among the individual resources in the system does not affect performance because matchmaking and scheduling are performed on the resource slots, each of which has an equal speed of execution. As a result, system performance does not change if c_r is different for the different resources in R as long as the sum of the number of resource slots in all the resources in R remains the same. Thus, the *total resource capacity* of the system, the sum of the number of resource slots for all the resources in R, is an important parameter that can affect system performance. This parameter is varied by varying the value of m in the experiments described in Section "Effect of the Number of Resources".

The configuration of RM-DCWF is defined as x-y-z where x specifies the laxity distribution algorithm (i.e., PD or ED, described in Section "Deadline Budgeting Algorithm for Workflows"), y specifies the approach to calculate the laxity of the job (i.e., SL or TL, described in Section "Deadline Budgeting Algorithm for Workflows"), and z specifies the task scheduling policy (i.e., TSP1 or TSP2, described in Section "Job Mapping Algorithm"). In total, there are 8 different RM-DCWF

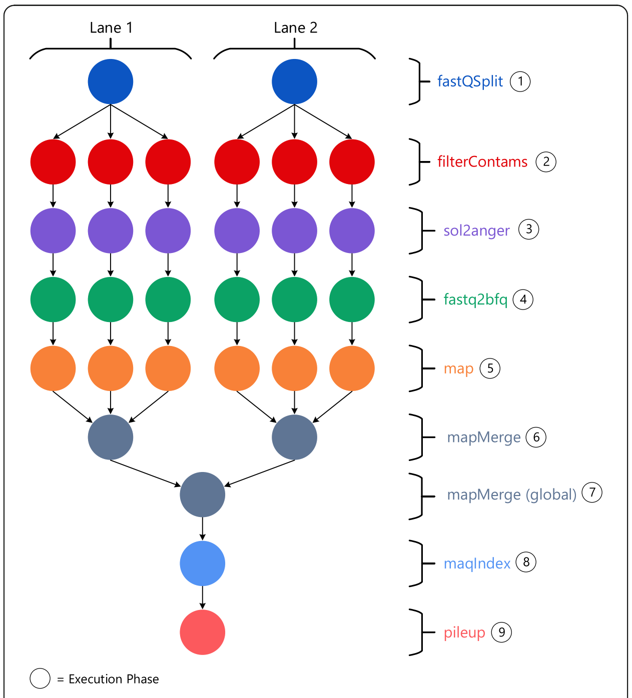

Fig. 5 DAG of a sample Genome application (based on [33])

Table 1 System and Workload Parameters for the Factor-at-a-Time Experiments

Parameter	Values	Default Value
Workload		
Type	{CyberShake, LIGO, Genome}	–
Job arrival rate (job/s)	λ_{CS} = {1/18, 1/22, 1/30} λ_{LG} = {1/150, 1/180, 1/265} λ_{GN} = {1/1800, 1/2290, 1/3205}	λ_{CS} = 1/22 λ_{LG} = 1/180 λ_{GN} = 1/2290
Earliest start time of jobs, s_j (sec)	$s_j = \begin{cases} at_j, & x = 0 \\ at_j + DU(1, s_{max}) & x = 1 \end{cases}$ where at_j is the arrival time of job j, $x \sim$ Bernoulli(0.5), and s_{max} = {1, 5, 25} * 10^4	s_{max} = 50,000
Job Deadline, d_j (sec)	$d_j = s_j + SET_j^R * em$ where $em \sim U(1, em_{max})$ and em_{max} = {2, 5, 10}	em_{max} = 5
System		
Number of Resources, m	m = {40, 50,60}	m = 50
Resource Capacity	c_r = 2	–
Configuration of RM-DCWF		
Laxity Distribution Algorithm	{PD, ED}	–
Approach to calculate the job laxity	{SL, TL}	–
Task Scheduling Policy	{TSP1, TSP2}	–

configurations, and thus, for each workload type, the factor-at-a-time experiments are conducted 8 times. This is performed to determine which configuration provides the best performance for a given workload.

Results of the factor-at-a-time experiments

The results of the factor-at-a-time experiments are presented in this section. Each simulation run was executed long enough to ensure that the system was operating at a steady state. Furthermore, each factor-at-a-time experiment is repeated a sufficient number of times such that the desired trade-off between simulation run length and accuracy of results was achieved. The confidence intervals for T and O at a confidence level of 95% are observed to remain less than ±5% of the respective average values in most cases. For P, the confidence intervals are observed to be in most cases less than ±10% of the average value. Such an accuracy of the simulation results is deemed adequate for the nature of the investigation: the focus of which is investigating the trend in the variation of a given performance metric in response to changes in the system and workload parameters and to compare the performance of the various RM-DCWF configurations. The values averaged over the simulation runs and the confidence intervals are shown in the figures and tables presented in this section. In the figures, the confidence intervals are shown as bars originating from the mean values; however, some of the bars are difficult to see since the confidence intervals are small. Note that the confidence intervals are considered while deriving a conclusion regarding the relative performance of the respective RM-DCWF configurations.

To conserve space and provide clarity of presentation, only the results of the two RM-DCWF configurations, one using PD and the other one using ED, that demonstrated the best overall performance in terms of P are presented in the following sub-sections. A more detailed discussion of the results of the performance evaluation can be found in [34]. More specifically, the two RM-DCWF configurations that are compared for each workload type are summarized:

- PD-SL-TSP1 vs ED-SL-TSP2 for the CyberShake workload
- PD-SL-TSP1 vs ED-SL-TSP1 for the LIGO workload
- PD-SL-TSP1 vs ED-SL-TSP1 for the Genome workload

Note that in the following sub-sections, the results of the experiments using the CyberShake workload are shown in figures, where P is displayed in its own figure and T and O are graphed in the same figure with T being displayed as a bar graph that uses the scale on the left Y-axis and O being displayed as a sequence of points that uses the scale on the right Y-axis. To maintain a reasonable number of figures, the results of each of the experiments using the LIGO and Genome workloads are shown in their own tables where the values of P, T, and O can be presented concisely.

Effect of job arrival rate

The impact of the job arrival rate on system performance is discussed in this section. The results of the experiments using the CyberShake workload are presented in Fig. 6 and Fig. 7. The figures show that for PD-SL-TSP1, *P*, *T*, and *O* increase with λ_{CS}. When λ_{CS} is high, jobs arrive on the system at a faster rate, which leads to more jobs being present in the system at a given point in time and an increased contention for resources. This in turn prevents some jobs from executing at their earliest start times, resulting in *T* increasing and some jobs to miss their deadlines (which increases *P*). The increased contention for resources also causes *O* to increase because RM-DCWF takes more time to find a resource to map the tasks of the job such that the job does not miss its deadline. Furthermore, since jobs are more prone to miss their deadlines at high values of λ_{CS}, RM-DCWF's Job Remapping algorithm, which is a source of overhead, is invoked more often, contributing to the increase in *O*.

It is observed that for ED-SL-TSP2, *P* and *O* increase with λ_{CS}, and *T* tends to remain relatively stable. In addition, when λ_{CS} is 1/22 jobs per sec or lower, both systems achieve comparable values of *P*; however, when λ_{CS} is 1/18 jobs per sec, ED-SL-TSP2 is observed to achieve a lower *P*. This can be attributed to ED-SL-TSP2 efficiently using the laxity of jobs to delay the execution of jobs with a later deadline to execute jobs with an earlier deadline, which in turn reduces the contention for resources at certain points in time and leads to a lower *P*. Although, as shown in Fig. 7, by delaying the execution of jobs, ED-SL-TSP2 achieves a higher *T* compared to PD-SL-TSP1. The *O* of ED-SL-TSP2 is higher compared to that of PD-SL-TSP1 when λ_{CS} is 1/22 jobs per sec or smaller. This is because more time is required by TSP2 to search for a resource that can execute a task at its latest possible time such that its sub-deadline is satisfied, compared to the time required by TSP1 to find a resource to execute tasks at their earliest possible times. However, when λ_{CS} is 1/18 jobs per sec,

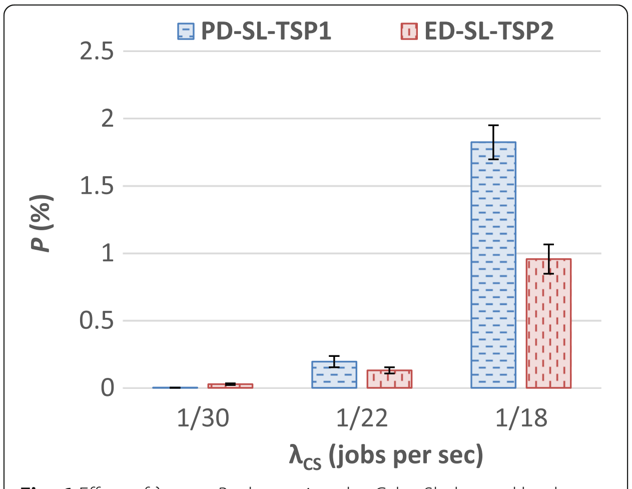

Fig. 6 Effect of λ_{CS} on *P* when using the CyberShake workload

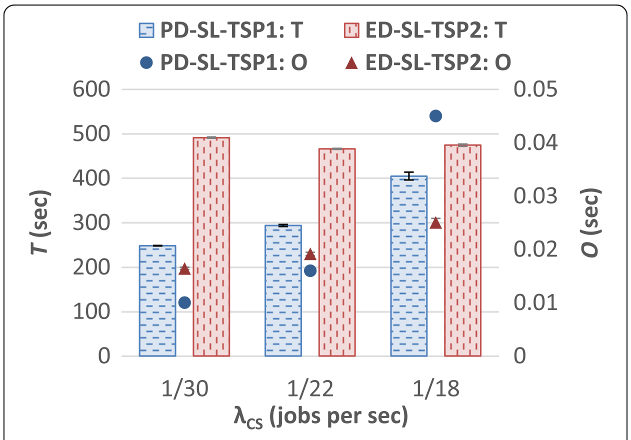

Fig. 7 Effect of λ_{CS} on *T* and *O* when using the CyberShake workload

PD-SL-TSP1 has a higher *O*, which can be attributed to the Job Remapping algorithm being invoked more often when using PD-SL-TSP1 compared to when using ED-SL-TSP2.

Table 2 and Table 3 present the results of the experiments when using the LIGO workload and the Genome workload, respectively. Unlike the CyberShake workload, when using the LIGO and Genome workloads, configuring RM-DCWF to use ED with TSP2 did not produce a better performance in comparison to using ED with TSP1. This demonstrates that TSP2 is only effective for certain workflows and the average job execution time and the structure of the job (e.g., precedence relationships between the tasks of the job) can affect the performance of TSP2. As shown in the tables, the trend in performance of *P*, *T*, and *O* are identical to that of the CyberShake workload when using PD-SL-TSP1. Furthermore, the results also show that both PD-SL-TSP1 and ED-SL-TSP1 achieve very similar results because TSP1 schedules tasks to start executing at their earliest possible times, regardless of their respective sub-deadlines. Over all the experiments performed to investigate the effect of the job arrival rate, the results demonstrate that RM-DCWF can achieve low values of *P* (less than 2% even at high arrival rates) and has a low

Table 2 LIGO workload: effect of λ_{LG} on *P*, *T*, and *O*

λ_{LG} (jobs/s)	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
1/265	0.02	0.02	1346	1346	0.008	0.008
	±0.01	±0.01	±0.6	±0.6	±0.00	±0.00
1/180	0.11	0.11	1466	1466	0.009	0.009
	±0.01	±0.01	±4.6	±4.6	±0.00	±0.00
1/150	1.03	1.06	2005	2006	0.017	0.016
	±0.12	±0.12	±29	±28	±0.001	±0.001

Table 3 Genome workload: effect of λ_{GN} on *P*, *T*, and *O*

λ_{GN} (jobs/s)	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
1/3205	0.01	0.01	17,544	17,544	0.008	0.008
	±0.00	±0.00	±927	±927	±0.000	±0.000
1/2290	0.07	0.07	17,963	17,963	0.008	0.008
	±0.01	±0.01	±1007	±1007	±0.000	±0.000
1/1800	1.43	1.40	52,312	52,472	0.048	0.051
	±0.45	±0.44	±12,915	±13,003	±0.015	±0.016

processing overhead as indicated by the small *O* (less than 0.025 s) and small *O/T* (less than 0.005%).

Effect of earliest start time of jobs

The impact of the earliest start time of jobs on system performance is described in this section. Fig. 8 and Fig. 9 present the results when using the CyberShake workload. It is observed that for PD-SL-TSP1, *P*, *T*, and *O* decrease with an increase in s_{max}. When s_{max} is large, jobs have a wider range of earliest start times with some jobs having an earliest start time near their arrival times, while other jobs have their earliest start times further in the future. This leads to less contention for resources and allows more jobs to execute at or closer to their earliest start times, resulting in a lower *P*, *T*, and *O*. Similar to PD-SL-TSP1, it is observed that for ED-SL-TSP2, *P* and *O* decrease as s_{max} increases. However, *T* is observed to increase with s_{max}. This is due to ED-SL-TSP2 scheduling tasks to execute at their latest possible times, while ensuring the respective sub-deadlines of the tasks are met. When the contention for resources is low (e.g., when s_{max} is large), ED-SL-TSP2 can more readily schedule tasks to start executing at their latest possible start times since jobs are less prone to miss their deadlines and the Job Remapping algorithm does not need to be invoked as often. Overall, it is observed that similar to the results presented in the previous section, ED-SL-TSP2 tends to achieve a lower *P* (35% lower on average), but this is accompanied by a higher *T* (75% higher on average) and higher *O* (32% higher on average) compared to PD-SL-TSP1.

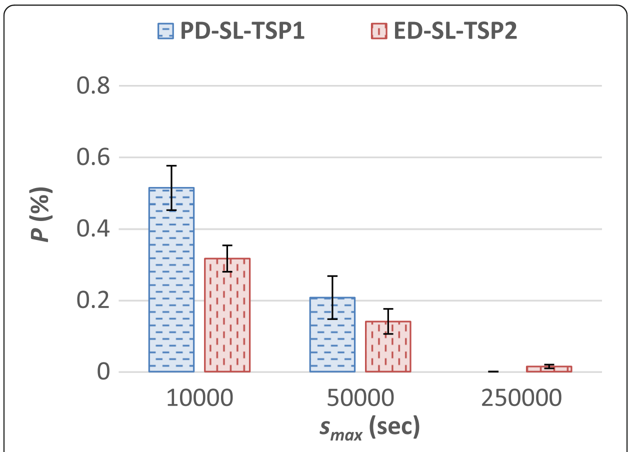

Fig. 8 Effect of s_{max} on *P* when using the CyberShake workload

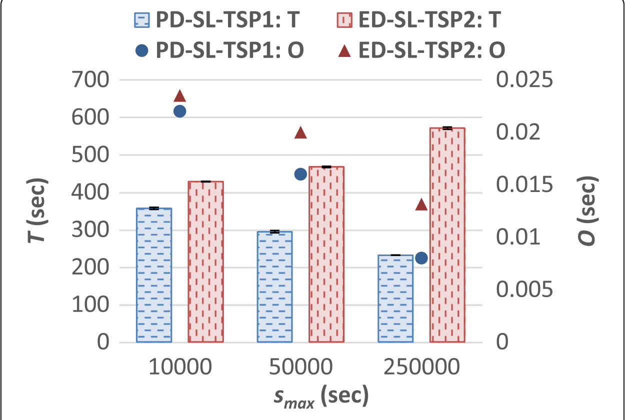

Fig. 9 Effect of s_{max} on *T* and *O* when using the CyberShake workload

The results of the experiments using the LIGO workload are presented in Table 4. It is observed that for both systems, *P*, *T*, and *O* seem to be insensitive to s_{max}, which is different from the results of PD-SL-TSP1 shown in Fig. 8 and Fig. 9, where *P*, *T*, and *O* are observed to decrease as s_{max} increases. The reason for this can be attributed to the LIGO workload comprising jobs with higher average execution times compared to those of the CyberShake workload, as well as the values of s_{max} used not significantly reducing the amount of jobs that have overlapping execution times (i.e., not reducing the contention for resources). The average job execution time (on a single resource) of the CyberShake workload (equal to 1551 s) is much smaller compared to that of the LIGO workload (13,300 s).

Table 5 presents the results of the experiments using the Genome workload. It is observed that *P* and *T* tend to increase and *O* remains stable as s_{max} increases. The increase in *P* could be attributed to the values of s_{max} experimented with (e.g., 50,000 and 250,000 s) causing more jobs to have overlapping execution times and thus increasing the contention for resources. This did not happen when using the other two workloads because the Genome workload comprises jobs with very high average execution times (~160,213 s on a single resource), which is significantly higher compared to those of the CyberShake and LIGO workloads. Increasing the values of s_{max} experimented with when using the Genome workload is expected to generate a similar trend in performance to the results of the CyberShake workload. This is because there will be less chance for the execution of jobs to overlap with one another.

Table 4 LIGO workload: effect of s_{max} on P, T, and O

s_{max} (sec)	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
10,000	0.10	0.10	1450	1450	0.009	0.009
	±0.01	±0.01	±3.3	±3.3	±0.000	±0.000
50,000	0.11	0.11	1466	1466	0.009	0.009
	±0.01	±0.01	±4.6	±4.6	±0.000	±0.000
250,000	0.09	0.08	1441	1427	0.009	0.009
	±0.01	±0.01	±4.7	±4.1	±0.000	±0.000

Effect of job deadlines

The impact of job deadlines on system performance is presented in this section. The results of the experiments using the CyberShake workload, as depicted in Fig. 10 and Fig. 11, show that for both systems, P decreases as em_{max} increases. This is because at a higher em_{max} jobs have more laxity and are thus less susceptible to miss their deadlines. Moreover, for ED-SL-TSP2, T is observed to increase as em_{max} increases. This can be attributed to jobs not having to execute at or close to their s_j to meet their deadlines when they have more slack time and the Job Remapping algorithm having to be executed less often. In addition, RM-DCWF may delay the execution of some jobs to allow a job with an earlier deadline to execute first. On the other hand, when em_{max} is small, jobs need to execute closer to their earliest start times and the Job Remapping algorithm is invoked when a job cannot be scheduled to meet its deadline. O is thus observed to increase for both systems, as em_{max} decreases because it leads to multiple invocations of the Job Remapping algorithm.

When comparing PD-SL-TSP1 and ED-SL-TSP2, it is observed that both systems perform comparably in terms of P when em_{max} is 5 or 10. However, when em_{max} is 2, it is observed that PD-SL-TSP1 achieves a smaller P compared to ED-SL-TSP2. This is because when the deadlines of the jobs are more stringent, jobs need to execute closer to their earliest start times to meet their deadlines, which agrees with the objective of TSP1 and not with the objective of TSP2, which schedules jobs to execute at their latest possible times. Similar to the results described in the previous sections, PD-SL-TSP1 also achieves a lower T and a lower or similar O compared to ED-SL-TSP2.

The results of the experiments using the LIGO workload and Genome workload are presented in Table 6 and Table 7, respectively. It is observed that the trend in performance observed for both systems when using the LIGO and Genome workloads are identical to that of the CyberShake workload when using PD-SL-TSP1: P decreases, O decreases, and T remains approximately at the same level as em_{max} increases. Overall, it is observed that RM-DCWF can achieve a low P (less than 4.2%) even when jobs have tight deadlines (i.e., em_{max} is 2). In addition, O is small (less than 0.03 s), and the processing overhead, as indicated by O/T, is less than 0.01% for all the experiments described in this sub-section.

Table 5 Genome workload: effect of s_{max} on P, T, and O

s_{max} (sec)	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
10,000	0.04	0.04	17,693	17,693	0.008	0.008
	±0.01	±0.01	±959	±959	±0.000	±0.000
50,000	0.07	0.07	17,963	17,963	0.008	0.008
	±0.01	±0.01	±1007	±1007	±0.000	±0.000
250,000	0.08	0.08	18,171	18,171	0.008	0.008
	±0.01	±0.02	±1049	±1049	±0.000	±0.000

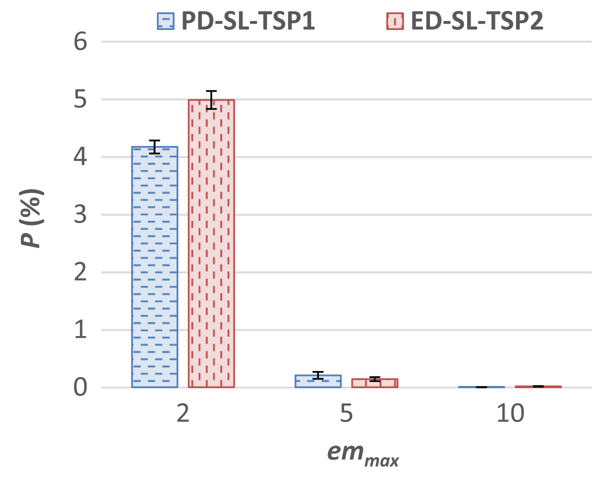

Fig. 10 Effect of em_{max} on P when using the CyberShake workload

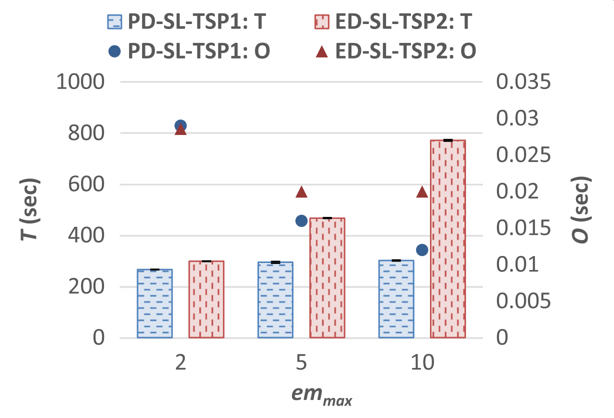

Fig. 11 Effect of em_{max} on T and O when using the CyberShake workload

Effect of the number of resources

In this section, the impact of m, the number of resources, on system performance is discussed. From the results of the experiments using the CyberShake workload (refer to Fig. 12 and Fig. 13), it is observed that for PD-SL-TSP1, P, T, and O decrease as m increases. This is because as m increases, there are more resources in the system to execute the jobs, leading to a lower contention for resources. The reason for the higher O when m is small can be attributed to the Job Mapping algorithm requiring more time to find a resource to map a task. When there are fewer resources in the system (small m), there are more tasks scheduled on each resource, leading to more time being required to find the ideal resource to execute a task. In addition, the high contention for resources makes jobs susceptible to miss their deadlines and leads to RM-DCWF's Job Remapping algorithm being invoked more often.

For ED-SL-TSP2, P, T, and O follow a similar trend in performance as observed for PD-TL-TSP1, except when m is 60. When m is 60, T is slightly higher compared to the case when m is 50. This can be attributed to the availability of ten additional resources, resulting in a lower contention for resources and a smaller P, and thus leading to a lower number of invocations of the Job Remapping Algorithm, which remaps jobs to start executing at their earliest possible times. This in turn allows TSP2 to schedule more tasks to execute at their latest possible times, while satisfying their respective sub-deadlines.

When comparing the performance of PD-SL-TSP1 and ED-SL-TSP2 for the CyberShake workload, it is observed that ED-SL-TSP2 achieves a smaller P and the most significant reduction in P is observed when m is 40 (refer to Fig. 12). Similar to the results presented in the previous sections (see Fig. 6, for example), scheduling tasks to execute at their latest possible time, while satisfying their respective sub-deadlines (i.e., using TSP2) tends to give rise to a lower P but a higher T when processing the CyberShake workload. The lower P can be attributed to ED-SL-TSP2 efficiently using the laxity of jobs to delay the execution of jobs with a later deadline to execute those with an earlier deadline. However, as shown in Fig. 13, it is observed that when m is 40, PD-SL-TSP1 achieves a higher T compared to ED-SL-TSP2. This can be attributed to PD-SL-TSP1 delaying the execution of multiple jobs that miss their deadlines for a long period of time for executing jobs that have not missed their deadlines. In the case of ED-SL-TSP2, fewer jobs need to be delayed because when m is 40, ED-SL-TSP2 achieves a smaller P compared to PD-SL-TSP1 (refer to Fig. 12).

The results of the experiments using the LIGO workload (see Table 8) and the Genome workload (see Table 9) follow a similar trend in system performance to that of the CyberShake workload when using PD-SL-TSP1: P decreases, T decreases, and O tends to decrease as m increases. It is observed once again that both PD-SL-TSP1 and ED-SL-TSP1 achieve similar results for both workloads. When m is 60, O is observed to be slightly higher compared to when m is 50. Even though, there is less contention for resources when m is 60, the Job Mapping algorithm may need to search through more resources to find the resource to schedule a task to start at its earliest possible time. This in turn leads to a slight increase in O.

Table 6 LIGO workload: effect of em_{max} on P, T, and O

em_{max}	P (%)		T (sec)		O (sec)	
	PD-S-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
2	2.44	2.43	1458	1457	0.012	0.011
	±0.14	±0.14	±4.4	±4.4	±0.000	±0.000
5	0.11	0.11	1466	1466	0.009	0.009
	±0.01	±0.01	±4.6	±4.6	±0.000	±0.000
10	0.04	0.04	1458	1463	0.009	0.008
	±0.01	±0.01	±6.2	±4.6	±0.000	±0.000

Table 7 Genome workload: effect of em_{max} on P, T, and O

em_{max}	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
2	0.49	0.49	17,933	17,933	0.009	0.009
	±0.12	±0.12	±1001	±1001	±0.000	±0.000
5	0.07	0.07	17,963	17,963	0.008	0.008
	±0.01	±0.01	±1007	±1007	±0.000	±0.000
10	0.03	0.03	17,963	17,963	0.007	0.007
	±0.01	±0.01	±1007	±1007	±0.000	±0.000

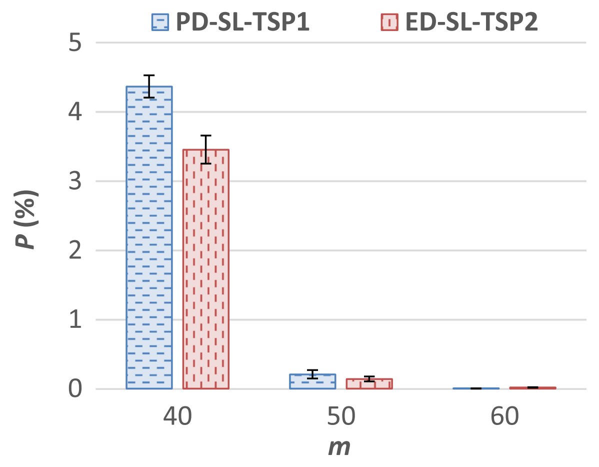

Fig. 12 Effect of m on P when using the CyberShake workload

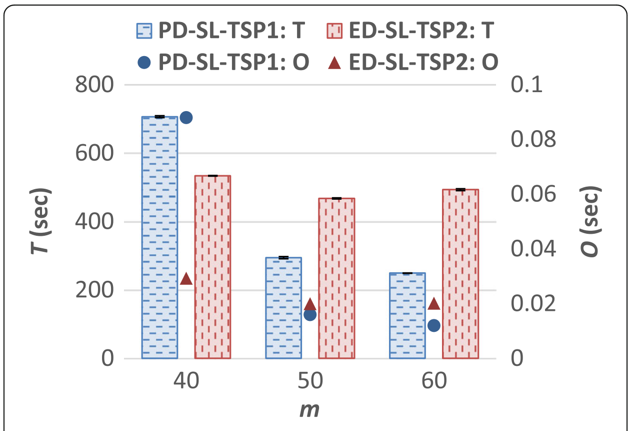

Fig. 13 Effect of *m* on *T* and *O* when using the CyberShake workload

Investigation of using a small number of resources

In the experiments described in Section "Effect of the Number of Resources", the number of resources, *m*, was varied from 40 to 60. This section describes the results of a set of experiments conducted to investigate the effect of using a small number of resources (20 resources, each with 2 resource slots). In this system, not all jobs will be able to execute at their maximum degree of parallelism. The fixed factors in this set of experiments include *m*, the workload, CyberShake, and all the other workload and system parameters described in Table 1 that are held at their default values. CyberShake is chosen because it is the workload that produced the most interesting set of results in the previous experiments in which the various configurations of RM-DCWF (captured in the last three rows of Table 1) exhibited different levels of performance. Since the number of resources is small, the default arrival rates for the CyberShake workload that are listed in Table 1 could not be used because it would lead to an unstable system (that is characterized by a job arrival rate exceeding the job service rate). Instead λ_{cs} is set to 1/45, 1/55, and 1/75 jobs per sec., corresponding to a resource utilization of approximately 0.9, 0.7, and 0.5. This is in line with the system utilizations achieved in the experiments described in Section "Results of the Factor-at-a-Time Experiments".

Table 8 LIGO workload: effect of *m* on *P*, *T*, and *O*

m	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
40	4.11	4.14	3210	3218	0.034	0.032
	±0.27	±0.27	±125	±126	±0.003	±0.003
50	0.11	0.11	1466	1466	0.009	0.009
	±0.01	±0.01	±4.6	±4.6	±0.000	±0.000
60	0.03	0.03	1360	1360	0.010	0.010
	±0.01	±0.01	±1.1	±1.1	±0.000	±0.000

Table 9 Genome workload: effect of *m* on *P*, *T*, and *O*

m	P (%)		T (sec)		O (sec)	
	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1	PD-SL-TSP1	ED-SL-TSP1
40	1.29	1.30	52,320	52,106	0.032	0.035
	±0.40	±0.42	±13,743	±13,597	±0.011	±0.012
50	0.07	0.07	17,963	17,963	0.008	0.008
	±0.01	±0.01	±1007	±1007	±0.000	±0.000
60	0.02	0.02	17,583	17,583	0.009	0.009
	±0.00	±0.00	±935	±935	±0.000	±0.000

Figures 14 and 15 show the effect of λ_{cs} on *P*, *T*, and *O* when processing the CyberShake workload on a system where *m* is 20. It is observed from Fig. 14 that PD-SL-TSP1 outperforms ED-SL-TSP2 in terms of *P*. This means that when the number of resources is small, it is more effective to schedule tasks to start executing at their earliest possible times (i.e., using TSP1) as opposed to scheduling tasks to execute at their latest possible times (using TSP2). In other words, when there are limited resources, the system should try to execute and complete jobs as soon as possible to lower the contention for resources when other jobs arrive on the system. Fig. 15 also shows that when λ_{cs} is 1/75 or 1/55 jobs per sec, PD-SL-TSP1 achieves a lower *T* and similar *O* compared to ED-SL-TSP2, once again demonstrating that for a small number of resources and low to moderate job arrival rates it is more effective to schedule tasks to execute at their earliest possible times. When λ_{cs} is 1/45 jobs per sec, PD-SL-TSP1 still achieves a lower *P* in comparison to ED-SL-TSP2. However, it is interesting to note that for this arrival rate that leads to a high contention for resources, PD-SL-TSP1 achieves a higher *T* and *O* in comparison to ED-SL-TSP2.

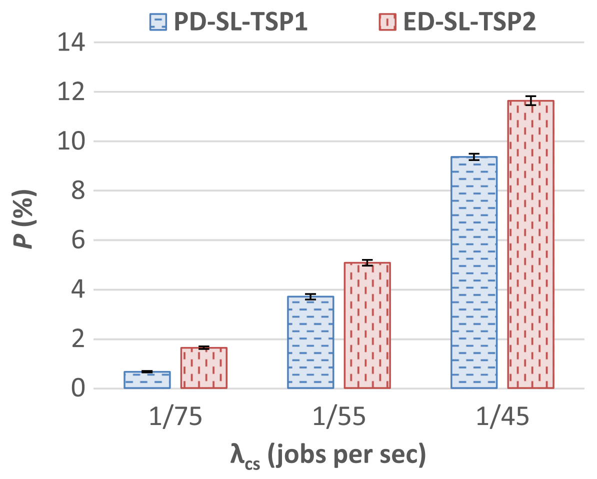

Fig. 14 Effect of λ_{cs} on *P* when using the CyberShake workload with *m* = 20

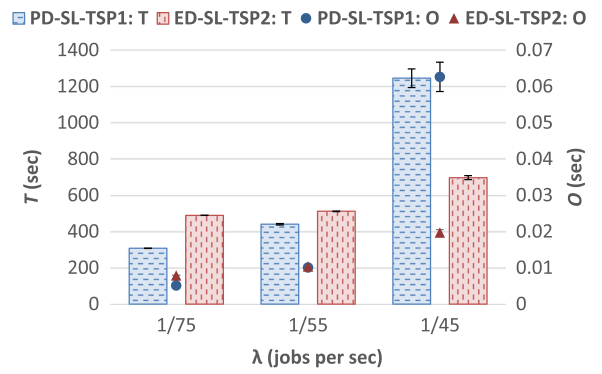

Fig. 15 Effect of λ_{cs} on *T* and *O* when using the CyberShake workload with *m* = 20

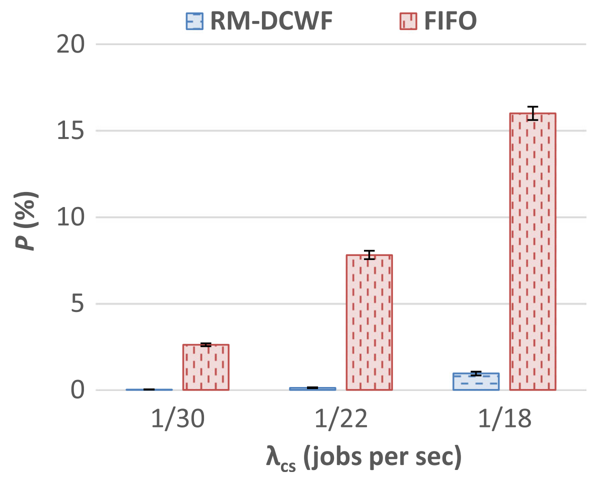

Fig. 16 RM-DCWF vs FIFO: effect of λ_{cs} on *P* when using the CyberShake workload

Comparison with a first-in-first-out (FIFO) scheduler

To the best of our knowledge there are no existing algorithms in the literature for matchmaking and scheduling on a cloud/cluster subjected to an open stream of workflows, each of which is characterized by general precedence relationships among its constituent tasks and a deadline for completion. In order to investigate the effectiveness of the various optimization techniques used in RM-DCWF, a comparison with a simple, conventional technique that does not use such optimization techniques is conducted. A first-in-first-out (FIFO) scheduling algorithm that can handle multi-phase workflows with general precedence relationships among its constituent tasks is devised for the purpose.

In this section, the results of experiments conducted to compare the performance of RM-DCWF with that of the conventional FIFO technique are presented. The FIFO technique devised is a simple scheduling strategy that stores jobs that arrive on the system in a FIFO queue. Thus, the jobs are scheduled in the same order in which they arrive on the system. As in the case with RM-DCWF, a best-fit strategy (as described in Section "Deadline Budgeting Algorithm for Workflows") is used allocate the tasks of a job to resources. For reasons similar to those discussed in Section "Investigation of Using a Small Number of Resources", the CyberShake workload is chosen as a fixed parameter and the job arrival rate, λ_{cs}, is varied. The remaining system and workload parameters (described in Section "System and Workload Parameters for the Factor-at-a-Time Experiments") are once again held at their default values. RM-DCWF is configured to use ED-SL-TSP2, which as described in Section "Effect of Job Arrival Rate", is observed to achieve the best performance when processing the CyberShake workload.

As shown in Fig. 16, RM-DCWF achieves a significantly lower *P* (99% to 94% smaller as λ_{cs} varies from 1/30 to 1/18 jobs per sec) in comparison to FIFO. This can be attributed to RM-DCWF prioritizing the execution of jobs with earlier deadlines, and FIFO executing jobs in the order they arrive on the system. Achieving a low *P* is the main objective for resource management performed by RM-DCWF. As far as the secondary performance metric *T* is concerned, FIFO achieves a lower *T* in comparison to RM-DCWF (see Fig. 17). This is expected because FIFO prioritizes executing jobs that have earlier arrival times and schedules each job to execute at its earliest possible time. This in turn allows jobs to finish executing earlier and results in jobs having a smaller turnaround time. Another reason for FIFO achieving a lower *T* is because RM-DCWF is configured to use TSP2, which schedules tasks to complete executing at their latest possible times, while not missing their sub-deadlines. Fig. 17 also displays that RM-WCDF achieves a slightly smaller *O* compared to FIFO. The higher *O* achieved by FIFO can be attributed to FIFO attempting to schedule all the arriving jobs, which

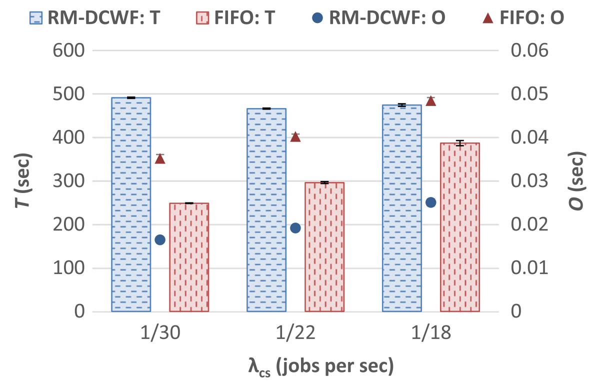

Fig. 17 RM-DCWF vs FIFO: effect of λ_{cs} on *T* and *O* when using the CyberShake workload

arrive relatively close to each other due to the high arrival rate, to execute at their earliest start times causing a high contention for resources. RM-WCDF makes use of the slack time of jobs to delay their execution, which in turn results in less contention for resources at certain points in time. The experimental results demonstrate the effectiveness of RM-DCWF that uses a number of techniques for optimizing system performance. This is demonstrated by the achieving of a lower *P* and *O* by RM-DCWF in comparison to a simple scheduling technique such as FIFO.

Comparison with MCRP-RM

This section discusses the results of the experiments conducted to compare the performance of RM-DCWF with that of MRCP-RM [29] (described in Section "Related Work"). Note that MRCP-RM is only applicable to jobs with two phases of execution, such as MapReduce [7] jobs, whereas in addition to MapReduce jobs, RM-DCWF can also handle jobs with different structures and more than two execution phases. Thus, the workload that is used in this comparison is a synthetic MapReduce workload that is used and described in [29]. RM-DCWF is configured to use PD-SL-TSP1, which is observed to have the best performance when processing this MapReduce workload. Factor-at-a-time experiments are performed to investigate the effect of various system and workload parameters on the performance of RM-DCWF and MRCP-RM.

Figure 18 and Table 10 present the performance of RM-DCWF and MRCP-RM in terms of *P*, *T*, and *O* as the job arrival rate (λ) is varied. As shown in Fig. 18, when λ is 0.0175 jobs per sec or smaller, the resource contention levels are low-to-moderate (e.g., average resource utilization is approximately less than 0.7), and it is observed that both RM-DCWF and MRCP-RM achieve comparable values of *P*. However, when λ is between 0.01875 to 0.0225 jobs per sec, generating a moderate-to-high contention for resources (e.g., average resource utilization is approximately between 0.7 and 0.85), MRCP-RM is observed to achieve up to a 22% lower *P* (on average 11% lower) compared to that achieved by RM-DCWF. At very high values of λ (e.g., 0.025 jobs per sec or higher), it is observed that the performance of MRCP-RM starts to deteriorate and RM-DCWF starts to outperform MRCP-RM (RM-DCWF has a 37% reduction in *P*). This can be attributed to the very high contention for resources (average resource utilization is greater than 0.9) causing jobs to queue up on the system and MRCP-RM having to solve complex constraint programs comprising a large number of decision variables and constraints. MRCP-RM requires more time to solve these complex constraint programs, which results in *O* increasing. The high *O* causes a delay in the execution of jobs and leads to jobs missing their deadlines. For all the values of λ experimented with, it is observed that RM-DCWF achieves a significantly lower *O* compared to MRCP-RM (on average 85% lower) (see Table 10). This can be attributed to RM-DCWF using a heuristic-based matchmaking and scheduling algorithm that is less computationally-intensive compared to MRCP-RM's matchmaking and scheduling algorithm, which is based on constraint programming.

The results of the other factor-at-a-time experiments performed to compare RM-DCWF and MRCP-RM, which can be found in [34], demonstrate a relative performance achieved by RM-DCWF and MRCP-RM that is similar to the results described in this section. When the contention for resources is reasonable, MRCP-RM and RM-DCWF achieve comparable values of *P* and *T*. On the other hand, when the contention for resources is high, such as when λ is 0.025 job per sec, RM-DCWF is observed to achieve a lower *P* compared to MRCP-RM. At these higher system loads, RM-DCWF achieves a

Table 10 RM-DCWF vs MRCP-RM: effect of λ on *T* and *O*

λ (jobs/s)	T (sec)		O (sec)	
	RM-DCWF	MRCP-RM	RM-DCWF	MRCP-RM
0.001	383 ± 2	383 ± 2	0.010 ± 0	0.018 ± 0
0.01	395 ± 2	394 ± 2	0.010 ± 0	0.043 ± 0
0.015	413 ± 2	417 ± 2	0.010 ± 0	0.074 ± 0
0.175	429 ± 2	435 ± 2.5	0.012 ± 0	0.10 ± 0
0.1875	444 ± 3	450 ± 3	0.013 ± 0	0.12 ± 0
0.02	465 ± 4	471 ± 3	0.012 ± 0	0.18 ± 0.01
0.02125	502 ± 6	508 ± 6	0.016 ± 0	0.25 ± 0.02
0.0225	564 ± 12	566 ± 10	0.019 ± 0	0.43 ± 0.04
0.025	1332 ± 76	1788 ± 118	0.110 ± 0	2.9 ± 0.48

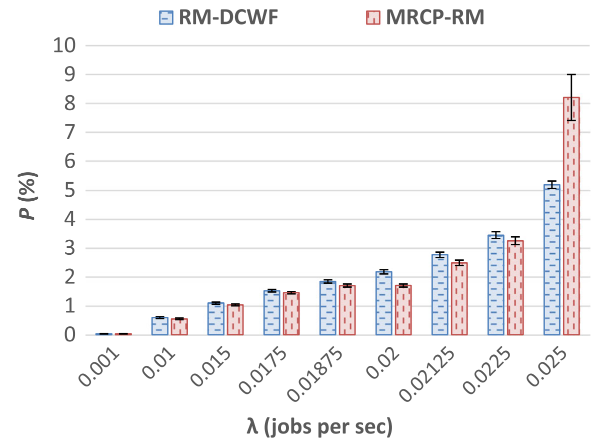

Fig. 18 RM-DCWF vs MRCP-RM: effect of λ on *P*

superior performance and thus demonstrates a higher scalability when system load is high.

Conclusions and future work

This paper describes a resource allocation and scheduling technique called RM-DCWF that can efficiently perform matchmaking and scheduling for an open stream of multi-stage jobs (workflows) with SLAs on a computing environment such as a private cluster or a set of resources acquired a priori from a public cloud. Each job arriving on the system is characterized by a SLA comprising an earliest start time, an execution time, and an end-to-end deadline. The RM-DCWF algorithm decomposes the end-to-end deadline of a job into sub-deadlines, each of which is associated with a task in the job. The individual tasks of the job are then mapped on to the resources where the objective is to satisfy the job's deadline and minimize the number of late jobs in the system. An in-depth simulation-based performance evaluation is conducted to investigate the effectiveness of RM-DCWF. The workloads used in the experiments are based on real scientific workflows from various fields of study, including biology and physics. A number of insights into system behaviour and performance are gained by analyzing the experimental results and are summarized next.

- *Effect of system and workload parameters*: An increase in the job arrival rate, λ, or a decrease in the earliest start time of jobs, s_j, or a decrease in the deadline of jobs, d_j, or a decrease in the number of resources, m, tends to lead to an increase in the proportion of late jobs, P, due to the increased contention for resources.
- *RM-DCWF configuration using the Proportional Distribution of Job Laxity Algorithm (PD)*: Overall, it is observed that using Task Scheduling Policy 1 (TSP1) generates lower or similar values for the proportion of late jobs, P, average job turnaround time, T, and average job matchmaking and scheduling time, O, than those achieved with Task Scheduling Policy 2 (TSP2). Furthermore, the two approaches used to calculate the laxity of the job (Sample Laxity, SL and True Laxity, TL) achieve similar performance with the SL approach achieving a slightly smaller P in most cases. When using PD, the results of the experiments showed that the highest performing RM-DCWF configuration (in terms of P) for all three workloads experimented with is *PD-SL-TSP1*.
- *RM-DCWF configuration using the Even Distribution of Job Laxity Algorithm (ED)*: The results demonstrate that for the CyberShake workload using ED-SL-TSP2 achieves the lowest P in most cases. However, when using the LIGO and Genome workloads, the best performance in terms of P is achieved by ED-SL-TSP1. When using ED, the results of the experiments showed that the two approaches used to calculate the laxity of the jobs (SL and TL) achieve comparable performance.
- *PD-based configuration* vs *ED-based configuration*: For the CyberShake workload, it is observed that overall ED-SL-TSP2 outperforms PD-SL-TSP1 in terms of P but it has a slightly higher T because TSP2 schedules tasks to execute at their latest possible times while meeting their respective sub-deadlines. In the case of the LIGO and Genome workloads, both PD-SL-TSP1 and ED-SL-TSP1 achieve similar values of P, T, and O. This can be attributed to TSP1 scheduling tasks to execute at their earliest possible times, regardless of their sub-deadlines.
- *Effectiveness of RM-DCWF*: For the system and workload parameters experimented with in Section "Results of the Factor-at-a-Time Experiments", it is observed that RM-DCWF can achieve low values of P (on average 0.62%). Even when the contention for resources is high and jobs are more susceptible to miss their deadlines (e.g., when λ is high, or d_j is small, or m is small), P is less than 5% and on average 2.2% for all the experiments conducted.
- *Efficiency of RM-DCWF*: Over all the experiments described in Section "Results of the Factor-at-a-Time Experiments", RM-DCWF achieved low values of O (less than 0.05 s and on average 0.02 s). Furthermore, O/T, an indication of the matchmaking and scheduling overhead, is also very small (less than 0.01%) for all the experiments conducted.
- *Effect of using a small number of resources*: When executing the CyberShake workload on a system with $m = 20$, which prevented some jobs from executing at their maximum degree of parallelism, it was observed that PD-SL-TSP1 outperforms ED-SL-TSP2 in terms of P. When there are a limited number of resources, better performance can be achieved if the system schedules jobs to complete executing as soon as possible to lower the contention for resources when other jobs arrive later on the system.
- *Comparison with the FIFO Scheduler:* The results of experiments comparing RM-DCWF with a FIFO Scheduler demonstrated that RM-DCWF achieves a significantly lower P and a lower O compared to the FIFO Scheduler. However, as expected the FIFO Scheduler did achieve a lower T because it prioritizes executing jobs that have earlier arrival times and schedules each job to execute at its earliest possible time.

- *MapReduce Workload and Comparison with MRCP-RM*: A summary of observations resulting from the performance evaluation to compare RM-DCWF and MRCP-RM when using a MapReduce workload is provided. At low-to-moderate contention for resources (e.g., $\lambda \leq 0.0175$ jobs per sec), RM-DCWF and MRCP-RM achieve comparable performance in terms of *P*. When the contention for resources is moderately high (e.g., average resource utilization is approximately 0.8), for a λ of 0.02 jobs per sec, for example, MRCP-RM outperforms RM-DCWF. At very high contention for resources (e.g., $\lambda \geq 0.025$ jobs per sec), RM-DCWF outperforms MRCP-RM. For all the values of λ experimented with, RM-DCWF achieves on average an *O* that is 85% lower compared to that achieved by MRCP-RM.
- These observations indicate that MRCP-RM's constraint programming-based resource management algorithm led to its superior performance over the heuristic-based RM-DCWF at medium system load, but because of its lower overhead RM-DCWF demonstrated higher scalability and superior performance at higher system loads.

Overall, the results of the experiments demonstrate that the objective of the paper that concerns the devising of an effective resource allocation and scheduling technique for processing an open stream of multi-stage jobs with SLAs on a cluster or a cloud with a fixed number of resources has been realized. RM-DCWF demonstrated that it can generate a schedule leading to a small *P* and *T* with a small *O* and *O/T* over a wide range of workload and system parameters experimented with. The choice of which RM-DCWF configuration (laxity distribution algorithm and task scheduling policy) to use is dependent on the workload to process; however, a good starting point is to use PD-SL-TSP1. If the system does not exhibit satisfactory performance, RM-DCWF can be reconfigured to use ED-SL-TSP2 when the next job arrives on the system or the next time this the same type of workload needs to be processed. When using TSP1, the choice of whether to use PD or ED, and SL or TL is not crucial as all the configurations using TSP1 achieve similar performance. However, if TSP2 is used, it is observed that using ED-SL-TSP2 typically achieves better performance compared to the other configurations that include TSP2.

A direction for future work is to adapt the resource management techniques to work in a computing environment where the number of resources in the system can be dynamically changed. Moreover, the resource management techniques can also be adapted to distributed computing environments with heterogeneous resources and multi-datacentre environments. This can involve devising more advanced techniques for supporting data locality when processing multi-stage jobs, which includes techniques for estimating the data transmission time and processing time for tasks based on the input data size and networking/processing capacities of the resources. Supporting data locality for multi-stage jobs that are characterized by multiple phases of execution may need to consider the possibility of one phase of execution sharing data with another phase of execution. If data needs to be shared among these two phases of execution, the tasks in these two phases of execution should be assigned to execute on nodes that are as close to each other as possible to minimize the data transmission overhead. Lastly, validating the results of the experiments for additional cases by conducting experiments using different combinations of workload and system parameters is also a part of the plan for future research.

Abbreviations
DAG: Directed Acyclic Graph; DBW: Deadline Budgeting Algorithm for Workflows; ED: Even Distribution of Job Laxity Algorithm; MRCP-RM: MapReduce Constraint Programming based Resource Management technique; PD: Proportional Distribution of Job Laxity Algorithm; RM-DCWF: Resource Management Technique for Deadline-constrained Workflows; SL: Sample Laxity; SLA: Service Level Agreement; TL: True Laxity; TSP1: Task Scheduling Policy 1; TSP2: Task Scheduling Policy 2

Acknowledgments
We are grateful to Huawei Technologies Canada and the Natural Sciences and Engineering Research Council of Canada (NSERC) for supporting this research.

Authors' contributions
The work presented in this paper is based on NL's Ph.D. research and thesis, which is supervised by SM. PAS is a collaborator and industrial partner for this research. NL devised and implemented the algorithms and conducted the simulation experiments. SM and PAS provided guidance and participated in system design. All the authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada. [2]Huawei Technologies Canada, Kanata, ON, Canada.

Received: 27 January 2017 Accepted: 4 September 2017
Published online: 29 September 2017

References

1. Columbus L (2015) Roundup Of Cloud Computing Forecasts And Market Estimates, 2015. http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/. Accessed 14 Nov 2016
2. Gartner (2015) Gartner Says Worldwide Cloud Infrastructure-as-a-Service Spending to Grow 32.8 Percent in 2015. http://www.gartner.com/newsroom/id/3055225. Accessed 14 Nov 2016
3. Manvi SS, Shyam GK (2013) Resource management for infrastructure as a service (IaaS) in cloud computing: a survey. Journal of Network and Computing Applications 41:424–440
4. Buyya R, Garg SK, Calheiros RN (2011) SLA-oriented resource provisioning for cloud computing: challenges, architecture, and solutions. In: proceedings

of the international conference on cloud and service computing, Hong Kong, China, 12-14 Dec 2011, p 1-10
5. Foster I, Kesselman C, Lee C, Lindell B, Nahrstedt K, Roy (1999) a distributed resource management architecture that supports advance reservations and co-allocation. In: Proceedings of the International Workshop on Quality of Service, London, UK, 1-4 June 1999, p 27–36
6. Maheswaran M, Siegel HJ (1998) A dynamic matching and scheduling algorithm for heterogeneous computing systems. In: proceedings of the heterogeneous computing workshop, Orlando, USA, 30 march 1998, p 57-69
7. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: proceedings of the international symposium on operating system design and implementation, San Francisco, USA, 6-8 Dec 2004, p 137-150
8. Dittrich J, Quiane-Ruiz J-A (2012) Efficient Big Data Processing in HadoopMapReduce. VLDB 2012/PVLDB, 5:12:2014–2015
9. Collins M (2011) Hadoop and MapReduce: big data analytics. Gartner. Available: https://www.gartner.com/doc/1521016/hadoop-mapreduce-big-data-analytics. Accessed 13 Jan 2017
10. Gift N (2010) Solve cloud-related big data problems with MapReduce. IBM. Available: http://www.ibm.com/developerworks/cloud/library/cl-bigdata/. Accessed 13 Jan 2017
11. Lim N, Majumdar S, Ashwood-Smith P (2014) Resource management techniques for handling requests with service level agreements. In: proceedings of the international symposium on performance evaluation of computer and telecommunication systems, Monterey, USA, 6-10 July 2014, p 618-625
12. Yu J, Buyya R, Tham CK (2005) Cost-based scheduling of scientific workflow applications on utility grids. In: proceedings of the international conference on e-science and grid computing, Melbourne, Australia, 5-8 Dec 2005, p 140-147
13. Abrishami S, Naghibzadeh M, Epema DHJ (2012) Cost-driven scheduling of grid workflows using partial critical paths. IEEE Transactions on Parallel Distributed Systems 23(8):1400–1414
14. Wan C, Wang C, Pei J (2012) A QoS-awared scientific workflow scheduling schema in cloud computing. In: proceedings of the international conference on information science and technology, Wuhan, China, 23-25 march 2012, p 634-639
15. Pandey S, Wu L, Guru S, Buyya R (2010) A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In: proceedings of the international conference on advanced information networking and applications, Perth, Australia, 20-23 April 2010, p 400-407
16. Chen WN, Zhang J (2012) A set-based discrete PSO for cloud workflow scheduling with user-defined QoS constraints. In: proceedings of the international conference on systems, man, and cybernetics, Seoul, South Korea, 14-17 Oct 2012, p 773-778
17. Bilgaiyan S, Sagnika S, Das M (2014) Workflow scheduling in cloud computing environment using cat swarm optimization. In: proceedings of the international advance computing conference, Gurgaon, India, 21-22 Feb 2014, p 680-685
18. Bittencourt LF, Senna CR, Madeira ERM (2010) Scheduling service workflows for cost optimization in hybrid clouds. In: proceedings of the international conference on network and service management, Niagara Falls, Canada, 25-29 Oct 2010, p 394-397
19. Szabo C, Kroeger T (2012) Evolving multi-objective strategies for task allocation of scientific workflows on public clouds. In: proceedings of the IEEE congress on evolutionary computation, Brisbane, Australia, 10-15 June 2012, p 1-8
20. Zhu Z, Zhang G, Li M, Liu X (2016) Evolutionary multi-objective workflow scheduling in cloud. IEEE Transactions on Parallel and Distributed Systems 27(5):1344–1357
21. Goudarzi H, Pedram M (2011) Multi-dimensional SLA-based resource allocation for multi-tier cloud computing systems. In: proceedings of the international conference on cloud computing, Washington, USA, 4-9 July 2011, p 324-331
22. Meng X, Lizhen C, Haiyang W, Yanbing B (2009) A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing. In: proceedings of the international symposium on parallel and distributed processing with applications, Chengdu and Jiuzhai Valley, China, 10-12 Aug 2009, p 629-634
23. Li X, Qian L, Ruiz R (2016) Cloud workflow scheduling with deadlines and time slot availability. IEEE Transactions on Services Computing, Preprint available: http://ieeexplore.ieee.org/document/7383307/
24. Verma A, Cherkasova L, Kumar VS, Campbell RH (2012) Deadline-based workload management for MapReduce environments: pieces of the performance puzzle. In: proceedings of the network operations and management symposium, Maui, Hawaii, USA, 16-20 April 2012, p 900-905
25. Mattess M, Calheiros RN, Buyya R (2013) Scaling MapReduce applications across hybrid clouds to meet soft deadlines. In: proceedings of the international conference on advanced information networking and applications, Barcelona, Spain, 25-28 march 2013, p 629-636
26. Hwang E, Kim KH (2012) Minimizing cost of virtual Machines for Deadline-Constrained MapReduce applications in the cloud. In: proceedings of the international conference on grid computing, Beijing, China, 20-23 sept 2012, p 130-138
27. Lai ZR, Chang CW, Liu X, Kuo TW, Hsiu PC (2014) Deadline-aware load balancing for MapReduce. In: proceedings of the international conference on embedded and real-time computing systems and applications, Chongqing, China, 20-22 Aug 2014, p 1-10
28. Chen C, Lin J, Kuo S (2015) MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems. IEEE Transactions on Cloud Computing, Preprint available: http://ieeexplore.ieee.org/document/7229311/
29. Lim N, Majumdar S, Ashwood-Smith P (2014) A Constraint Programming-Based Resource Management Technique for Processing MapReduce Jobs with SLAs on Clouds," In: Proceedings of the International Conference on Parallel Processing, Minneapolis, USA, 9-12 Sept 2014, p 411–421
30. Lim N, Majumdar S, Ashwood-Smith P (2017) MRCP-RM: a technique for resource allocation and scheduling of MapReduce jobs with deadlines. IEEE Transactions on Parallel and Distributed Systems 28(5):1375–1389
31. The Apache Software Foundation. Hadoop. http://hadoop.apache.org. Accessed 16 Jan 2016
32. Oracle Corporation. System.nanoTime(). https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime(). Accessed 14 Nov 2016
33. Bharathi S, Chervenak A, Deelman E, Mehta G, Su MH, Vahi K (2008) Characterization of scientific workflows. In: proceedings of the workshop on workflows in support of large scale science, Austin, USA, 17 Nov 2008, p 1-10
34. Lim N (2016) Resource management techniques for multi-stage jobs with deadlines running on clouds. Dissertation, Carleton University, Ottawa, ON, Canada, Ph.D