Resource Management Techniques for Multi-Stage Jobs with Deadlines Running on Clouds

by

Norman Lim, M.A.Sc. (ECE), B. Eng. (CSE)

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract

The objective of this thesis is to devise effective resource management techniques for processing a workload comprising an open stream of *multi-stage jobs* on a distributed computing environment, such as a private cluster or a set of resources acquired a priori from a public cloud. Each job is associated with a *service level agreement* (SLA*)* characterized by an earliest start time, an execution time, and an end-to-end deadline. The two important operations in resource management that this thesis focuses on are resource allocation (*matchmaking*) and *scheduling*. Given a pool of jobs to execute, a matchmaking algorithm chooses the resources to be allocated to a given job, whereas a scheduling algorithm determines the order in which these jobs are to be executed for achieving the desired system objectives. Multi-stage jobs have multiple phases of execution and require processing from multiple system resources.

Resource management techniques for processing *MapReduce* jobs (used for facilitating Big Data analytics), as well as other multi-stage jobs such as scientific workflows characterized by multiple phases of execution and different types of precedence relationships, are considered. The key techniques devised in this thesis research include: (1) resource management techniques based on decomposing the end-to-end deadline of a multi-stage job into components (e.g., sub-deadlines) each of which is associated with a specific task in the job, (2) alternative resource management algorithms that use optimization techniques, and (3) techniques for handling inaccuracy/error in user estimates of job execution times (submitted as part of the SLA). A comprehensive and rigorous performance evaluation of the resource management techniques is conducted using prototyping and measurement on a real system deployed on a cloud as well as simulation.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my thesis supervisor, Professor Shikharesh Majumdar, for his guidance, encouragement, and continuous support throughout the completion of my research and this thesis. I am greatly appreciative for his professionalism and the time and effort he invested through out my studies, including reviewing my research papers and giving me suggestions to improve my research.

In addition, I am grateful to Carleton University, the Government of Ontario, the Natural Science and Engineering Council (NSERC), and Huawei Technologies Canada for providing financial support for this research, which allowed me to fully focus my time and effort on my studies. I would also like to acknowledge Peter Ashwood-Smith from Huawei Technologies Canada for his support of this research.

Last but not least, I want to thank my parents, my brother, my sister, and my friends for their unwavering support and encouragement. I want to especially thank them for always believing in me and providing me motivation through some tough times. Without them, completing this thesis would not have been possible.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

λ	Job arrival rate
$\tau_{j,k}$	k^{th} task of job j
a_t	Assigned (or scheduled) start time of task t (decision variable used by CP Model)
at_j	Arrival time of job j
AT	A set that contains all the tasks of all the jobs in J
c_r^{mp}	Map task capacity of resource r
c_r^{rd}	Reduce task capacity of resource r
C	Completion time of a batch workload
CL_t	Cumulative laxity of task t
CL_j^{ph}	Cumulative laxity of execution phase ph for a job j
CT_j	Completion time of job j
d_j	Deadline of job j
DU	Discrete uniform distribution
e_t	Execution time of task t
e_t^{adj}	Adjusted execution time of task t
e_t^{est}	Estimated execution time of task t
e_t^{run}	Actual runtime of task t
$e_t^{threshold}$	Task execution time threshold used by Feitelson's Error Model
em	Execution time multiplier
em_{max}	Upper-bound of the uniform distribution used to generate em
eps_t	Earliest possible start time of task t
f	Execution time error factor
f_t	Execution time error factor of task t

I	Set of discrete time values (used by MILP Model)
J	Set of jobs
k_j^{mp}	Number of map tasks in job j
k_j^{rd}	Number of reduce tasks in job j
L_j	Laxity of job j
L_j^{act}	Actual laxity of job j
L_j^{ep}	Laxity for each execution phase in job j
L_j^{est}	Estimated laxity of job j
LN	Lognormal Distribution
LT_t	Laxity of task t
LT_t^{max}	Maximum laxity of task t
LT_t^{min}	Minimum laxity of task t
m	Number of resources in R
me	Map task execution time
me_{max}	Upper-bound of the discrete uniform distribution used to generate me
n	Number of jobs processed in an experiment
n_j^{ep}	Number of execution phases in job j
N	Number of jobs that miss their deadlines
N_j	Binary variable that is set to 1 if job j misses its deadline (used by MILP Model and CP Model)
NL_j	Normalized laxity of job j
o_j	The matchmaking and scheduling time of job j
O	Average job matchmaking and scheduling time
p	Probability that a job j has s_j greater than at_j
P	Proportion of jobs that miss their deadlines
PO	Processing time overhead of the solver for matchmaking and scheduling a batch workload

q_t	Resource capacity requirement of task *t*
re	Reduce task execution time
R	Set of resources that represent the distributed computing environment (comprises *m* resources)
s_j	Earliest start time (or release time) of job *j*
s_{max}	Upper-bound of the discrete uniform distribution used to generate the value that is added to at_j for calculating s_j of jobs that have s_j greater than at_j
sd_t	Sub-deadline of task *t*
sd_t^{LFPT}	Sub-deadline of the latest finishing parent task of task *t*
SCT_j	Sample completion time of job *j*
SET_j	Sample execution time of job *j*
SET_j^{max}	Maximum execution time of job *j*
SET_j^{min}	Minimum execution time of job *j*
SET_j^{R}	The execution time of job *j* when it executes at its maximum degree of parallelism on *R*
$SET_j^{R\_PL}$	The execution time of job *j* when it executes on *R,* while considering the current processing load of the resources in *R*.
ts_t	Earliest start time of task *t*
T	Average job turnaround time
T_j^{mp}	Set of map tasks in job *j*
T_j^{rd}	Set of reduce tasks in job *j*
U	Uniform distribution
x_{tr}	Binary variable that is set to true if task *t* is assigned on resource *r* (used by CP Model)
x_{tri}	Binary variable that is set to true if task *t* is assigned to execute on resource *r* at time *i* (used by MILP model)

Glossary of Terms

Amazon EC2	Amazon Elastic Cloud Compute
CP	Constraint Programming
CPLEX	IBM ILOG CPLEX Optimization Studio
CPS	CP-Scheduler
DAG	Directed acyclic graph
DBW	Deadline Budgeting Algorithm for Workflows
ED	Even Distribution of Job Laxity Algorithm
EDF	Earliest Deadline First
EDFS	EDF-Scheduler
ETIL	Hadoop's Eager Task Initialization Listener class
FIFO	First-in-first-out
HCP-RM	Hadoop Constraint Programming based Resource technique
HCP-RM-EH	HCP-RM with techniques to handle error in user-estimated execution times
HDFS	Hadoop Distributed File System
IaaS	Infrastructure-as-a-Service
IDE	Integrated Development Environment
IT	Information Technology
JIPL	Hadoop's Job In Progress Listener class
JQ-JIPL	Hadoop's Job Queue Job In Progress Listener class
JQTS	Hadoop's Job Queue Task Scheduler class
LFMT	Latest Finishing Map Task
LFRT	Latest Finishing Reduce Task
LIGO	Laser Interferometer Gravitational Wave Observatory
MILP	Mixed Integer Linear Programming

MinEDF-WC	Minimum Resource Quota Earliest Deadline First with Work-Conserving scheduling technique
MRCP-RM	MapReduce Constraint Programming based Resource Management technique
MRv1	Hadoop MapReduce Architecture Version 1
MRv2	Hadoop MapReduce Architecture Version 2
OPL	Optimization Programming Language
PaaS	Platform-as-a-Service
PD	Proportional Distribution of Job Laxity Algorithm
POpt1	Performance Optimization 1 for MRCP-RM: Separating the Matchmaking and Scheduling Operations
POpt2	Performance Optimization 2 for MRCP-RM: Handling Earliest Start Time of Jobs
PSEH	Prescheduling Error Handling technique
QoS	Quality of Service
RT	CP-Scheduler's REFERENCE_TIME variable
SaaS	Software-as-a-Service
setPTSubDL	WFBB-RM's *setParentTasksSubDeadline()* method
SimExec	Simulate Execution Time Hadoop application
SL	Sample Laxity
SLA	Service Level Agreement
TL	True Laxity
TSP1	WFBB-RM's Task Scheduling Policy 1
TSP2	WFBB-RM's Task Scheduling Policy 2
URL	Uniform Resource Locator
VM	Virtual Machine
WFBB-RM	Workflow Budget-Based Resource Management technique
XaaS	Everything-as-a-Service

Chapter 1 Introduction

Cloud computing has emerged as one of the most prevalent parallel and distributed computing paradigms. One of the key objectives of cloud computing is to deliver reliable services/applications hosted on the "cloud" such that consumers can access the services/applications anywhere and at any time (i.e., on-demand) via the Internet [1]. The authors of [1] define a "cloud" as follows:

"A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers."

The computer systems and resources of the cloud that include compute, storage, and network resources are often housed in a facility called a *datacenter*.

Since the emergence of cloud computing, its popularity has steadily increased and it is now deployed extensively in the domain of Information Technology (IT) because it improves and simplifies how IT is managed and consumed. A number of reputable financial institutions and market research organizations, including Merrill Lynch and Gartner, have predicted a multi-billion-dollar market for the cloud computing industry [2][3][4]. Furthermore, as pointed out in [5], the annual world-wide spending on cloud computing for enterprise IT is expected to increase significantly with time. Consequently, research on cloud computing is receiving a great deal of interest from researchers and practitioners from academia as well as industry.

The importance of cloud computing that provides *resources on demand* to various types of users, including enterprises and engineering and scientific institutions, is growing

rapidly [6]. Although cloud computing is an effective and economically viable solution for distributed computing, it poses a number of challenges. A survey [7] presented on cloud computing stated that *security* and *performance* are the two top priorities for cloud service consumers and cloud service providers. In a cloud, the user has a lack of control over the execution environment, and there is also a concern with the security and privacy of the data stored in the cloud. Another key issue in cloud computing is related to performance. Public cloud service providers, such as Amazon [8] and Microsoft [9], deploy datacenters that comprise a large pool of resources. Many enterprises and institutions also have their own private clouds for managing their IT infrastructure, performing data processing operations, and facilitating research. Irrespective of the type of cloud deployed, effective and efficient *resource management* techniques are crucial for harnessing the power of the underlying resource pool and to achieve the performance objectives of the system. Achieving a high system performance is important because it can lead to more satisfied users and high utilization of resources, leading to more revenue for the cloud service provider. For example, with an effective resource management technique the system can achieve a high job throughput, low job response times (latency), high utility of resources, and a high quality of service (QoS) for consumers [6]. Thus, the focus of this thesis is on the critical issue of resource management on clouds.

The rest of this chapter is organized as follows. In Section 1.1, the motivation and challenges behind the research of the thesis is described. Next, in Section 1.2, the objectives, contributions, and scope of the thesis are presented. Lastly, Section 1.3 provides an outline of how the remainder of the thesis is organized.

1.1 Motivations and Challenges for the Thesis

A significant amount of research is available in the area of resource management on parallel and distributed systems, including grids [10] (a predecessor of cloud computing) and clouds, characterized by on-demand jobs that are to be satisfied on a best effort basis. However, comparatively less work is available for resource management where jobs require a QoS that is often captured by a *service level agreement* (SLA). This is an active and open research area [1][6]. As in the case of grid computing [10], which also supported resources on demand, QoS and SLAs remain an important issue. A SLA, which defines a contract between the service requester and the service provider regarding the level of QoS associated with a job, is an important characteristic of cloud computing [11]. Ensuring that the SLA (i.e., QoS requirements) of a job is satisfied is important for service providers because it is imperative for achieving a high quality of experience and satisfaction for the cloud users. A satisfied user is more likely to use the same service provider again. Furthermore, violating a SLA can lead to fines for the service provider and a loss of revenue. The SLA may vary from application to application and various parameters can be included in a SLA, such as delays, packet loss, uptime, and mean time to recover. Similar to [1] and [11], this thesis focuses on SLAs that are characterized by an *earliest start time*, an *execution time*, and an end-to-end *deadline* for job completion. This, for example, is important for latency-sensitive business and scientific applications that require a timely processing of data. Note that the deadline for the job is a *soft deadline*, which means that the job can complete its execution even if it misses its deadline; however, the desired system objective is to minimize the number of such late jobs. Missing a deadline may lead to a low quality of experience for the user and a SLA violation for the service provider.

Thus, minimizing the number of late jobs, which leads to a higher number of satisfied users and a higher quality of service, is an important objective of the system.

Due to cloud computing becoming more prevalent, a variety of different applications are 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*). Most of the research on resource management for jobs characterized by SLAs have not considered:

1). Jobs requiring service from *multiple* system resources as required for supporting multi-stage jobs. Most of the research on resource management for jobs with SLAs has only considered jobs that need to be processed by a single resource.

2). An open stream of job arrivals. Most of the previous research in the literature has only addressed meeting the deadlines for a fixed number of jobs executing on the system (i.e., a batch workload). However, clouds are typically subjected to an open stream of job arrivals and not a fixed number of jobs.

The objective of the research presented in this thesis is aimed at filling this gap. The type of resources that this thesis is concerned with are *nodes* in a distributed system where each node has its own compute resources (CPU and memory), storage devices, and communication devices, as well as runs its own operating system. For example, each of these nodes can be PCs in a private cluster or alternatively, virtual machines (described in more detail in Section 2.1) provisioned from a public cloud.

Multi-stage jobs are important in a variety of contexts. For example, a job invoking an application with multiple components or a job comprising multiple *tasks* that require executing on multiple resources. Another scenario is in the context of a workflow that is

characterized by multiple phases of execution where each phase can comprise multiple tasks with precedence relationships and each phase requires execution on multiple devices. 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. When considering resource management for an open stream of multi-stage jobs with SLAs, a complex resource management problem arises that has a number of challenging issues that warrant further investigation. These challenges are summarized next.

- **SLA budgeting:** How to decompose the end-to-end SLA into components each of which is to be associated with a specific component in the end-to-end path traversed for processing the job?
- **Resource management algorithm:** Effective matchmaking and scheduling algorithms are required for selecting resources from a given resource pool and determining the order of execution of jobs mapped to the same resource for satisfying the SLAs. Matchmaking and scheduling is known to be a NP-hard problem, and when considering an open stream of multi-stage jobs with SLAs, the complexity of the problem increases significantly due to a continuous stream of jobs arriving on the system.
- **Single step resource management techniques**: Investigating alternate approaches that avoid the budget-based technique in order to make resource management decisions on multiple resources in a single joint step.

Note that a naïve solution to solve the resource management problem described is to reserve all the resources that a job requires for the entire duration of time specified in the SLA. However, this approach is not efficient because with multiple phases of execution,

each resource may not be used during each phase and reserving all the resources for the entire duration of time leads to poor resource utilization and less revenue for the service provider. Reserving each resource only for the duration it is used for executing a job is a more efficient and effective solution because it allows more jobs to be executed on the cloud, leading to a more efficient use of resources. Therefore, the techniques described in this thesis avoids the naïve solution and instead proposes more efficient and intelligent resource management techniques.

1.2 Objective and Contributions of the Thesis

The focus and objective of this thesis is to devise efficient resource management techniques and algorithms for processing an open stream of multi-stage jobs where each job is characterized by a SLA (comprising an earliest start time, an execution time, and an end-to-end deadline) on a parallel and distributed computing environment with a fixed number of resources (or nodes), such as a private cluster or a set of resources acquired a priori from a public cloud. Recall from the previous section that each resource (or node) has its own compute resources (CPU and memory), storage devices, and communication devices, as well as runs its own operating system. More specifically, the goal is to devise resource management techniques that can make decisions that minimize the number of jobs that miss their deadlines, while incurring a low processing overhead. Note that the resource management techniques that are presented in this thesis can be used by public cloud service providers such as Amazon, as well as by a user who uses his/her own resources that are available on a private cluster or that are acquired from a public cloud.

An example of a multi-stage application that this thesis considers is MapReduce [12] (described in more detail in Section 2.3), which is used by many companies and

institutions for processing and analyzing large data sets (i.e., facilitating *Big Data analytics*). Devising effective resource management techniques for processing MapReduce jobs with deadlines forms an important component of this thesis. In addition to MapReduce jobs, which are characterized by two phases of execution, this thesis also devises resource management techniques for processing workflows with different types of precedence relationships and more than two phases of execution, such as scientific workflows used in the domain of physics and biology.

The new resource management techniques and algorithms that are presented in this thesis have contributed to the state of the art in the field of resource management on clouds as reflected in the various research papers that have been published (described in more detail in Section 1.2.1). The major contributions of this thesis are summarized next:

- **Deadline budgeting algorithms:** Algorithms are devised to decompose the end-to-end deadline associated with a job into components (e.g., sub-deadlines) each of which is associated with a specific task in the job. In particular, algorithms are devised to handle both MapReduce type jobs characterized by two phases of executions and scientific workflows with different types of precedence relationships.

- **Budget-based resource management technique:** Effective matchmaking and scheduling algorithms are devised to process multi-stage jobs on parallel and distributed systems using the budgets (component SLAs) determined by the deadline budgeting algorithms.

- **Resource management techniques based on optimization methods:** Alternate resource management techniques, which avoid the budget-based technique, are

devised to make matchmaking and scheduling decisions on multiple resources in a single joint resource management step. These resource management techniques formulate and solve the resource management problem as an optimization problem using mixed integer linear programming (MILP) [13] and constraint programming (CP) [14]. Both MILP and CP are well-known theoretical techniques that can solve optimization problems and find optimal solutions. Various implementations of the formulations using commercial-off-the-shelf and open source software packages are considered, including IBM ILOG CPLEX Optimization Studio (CPLEX) [15].

  - Although previous works have used optimization methods to perform matchmaking and scheduling, most of these techniques are offline techniques and only work in a closed system for processing a batch workload with a fixed number of jobs. The resource management techniques based on optimization methods (and associated performance optimizations) that this thesis presents are devised to process an open stream of multi-stage jobs with SLAs.

- **Handling error associated with user-estimated job execution times:** Studies on real systems show that user estimates of job execution times (included in a SLA, for example) are error prone [16][17][18]. A novelty of the research presented in this thesis is the techniques devised for handling inaccuracy or error in user estimates of job execution times to build in the appropriate robustness into the resource management techniques. With little work existing in this area in the context of MapReduce type systems, addressing the challenges in devising

techniques to improve the robustness of the resource management algorithms makes a strong contribution to the state of the art.

- **Insights resulting from a performance analysis based on simulation and a proof-of-concept prototype:** An in-depth and rigorous performance evaluation of the resource management techniques is conducted using prototyping and measurement on a real system deployed on a cloud (Amazon EC2 [8]), as well as using simulation. A detailed analysis of the results is provided to gain insights into system behavior and performance, as well as to investigate the impact of various workload and system parameters on the performance of each technique. In addition, the relative performance of the resource management techniques compared to other techniques in the literature is provided. Lastly, a comparative performance analysis of the proposed techniques is performed to determine the system and workload parameters for which one technique is superior in performance to the others.

1.2.1 List of Publications

Based on the results of the research, four papers [19][20][21][22] have been published in refereed international conferences. In addition, a book chapter [23] and a journal article [24] have been accepted for publication. The details are presented next.

Archival Publications:

- N. Lim and S. Majumdar, "Resource Management for MapReduce Jobs Performing Big Data Analytics", in *Big Data Management, Architecture, and Processing*, K.-C. Li, H. Jiang, and A. Zomaya, Eds. USA: CRC Press, Taylor & Francis Group, 2016 (accepted for publication).

- N. Lim, S. Majumdar, and P. Ashwood-Smith, "MRCP-RM: a Technique for Resource Allocation and Scheduling of MapReduce Jobs with Deadlines", *IEEE Transactions on Parallel and Distributed Systems*, October 2016 (accepted for publication).

Refereed Conference Publications:

- N. Lim, S. Majumdar, and P. Ashwood-Smith, "Resource Management Techniques for Handling Requests with Service Level Agreements", *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Monterey, CA, USA, 6-10 July 2014, pp. 618 -625.

- N. Lim, S. Majumdar, and P. Ashwood-Smith, "Engineering Resource Management Middleware for Optimizing the Performance of Clouds Processing MapReduce Jobs with Deadlines", *International Conference on Performance Engineering (ICPE)*, Dublin, Ireland, 24 -26 March 2014, pp.161-172.

- N. Lim, S. Majumdar, and P. Ashwood-Smith, "A Constraint Programming-Based Resource Management Technique for Processing MapReduce Jobs with SLAs on Clouds", *International Conference on Parallel Processing (ICPP)*, Minneapolis, MN, USA, 9-12 Sept 2014, pp. 411-421.

- N. Lim, S. Majumdar, and P. Ashwood-Smith, "A Constraint Programming Based Hadoop Scheduler for Handling MapReduce Jobs with Deadlines on Clouds", *International Conference on Performance Engineering (ICPE)*, Austin, TX, USA, 31 Jan – 4 Feb 2015, pp. 111-122.

1.2.2 Scope of the Thesis

The resource management techniques described in this thesis are devised for distributed computing environments with a fixed number of resources (or nodes) subjected to a workload comprising an open stream of multi-stage jobs with SLAs. Such environments can include a cluster of computers or a set of virtual machines that are provisioned a priori from a cloud. Furthermore, the resource management techniques also consider a single datacentre environment where the network delays and data transmission times are negligibly small. Adapting the techniques for environments where the number of resources in the system can be dynamically changed or for geographically-dispersed multi-datacentre environments where the network delays and data transmission times may be large are out of scope for this research and can form a direction for future research, as described in Section 8.6.

As discussed earlier, the SLA submitted as part of the job includes an earliest start time, an execution time, and an end-to-end soft deadline. In this thesis, the estimated job execution times are provided by the user. The estimation of job execution times by the system instead of the user is beyond the scope of this thesis. However, this thesis does concern dealing with inaccuracy/errors in user-estimated job execution times, as described in Chapter 6. Moreover, the deadlines of the jobs submitted by the user are soft deadlines, which means that jobs are permitted to miss their deadlines but the objective of the system is to minimize the number of missed deadlines to preserve a high quality of service.

1.3 Outline of the Thesis

The remainder of the thesis is organized as follows. In Chapter 2, background information is provided and related work is discussed. The focus of Chapter 3 is on

describing the resource management techniques devised to process a batch of MapReduce jobs with SLAs. The MapReduce Constraint Programming based Resource Management (MRCP-RM) technique devised for systems subjected to an open stream of MapReduce jobs with SLAs, along with its adaption onto a real system, Hadoop [25], referred to as the Hadoop Constraint Programming based Resource Management (HCP-RM) technique, are discussed in Chapter 4 and Chapter 5, respectively. In Chapter 6, the techniques for improving the robustness of the resource management algorithms through handling of errors/inaccuracies in user-estimated execution times are discussed. The focus of Chapter 7 is on describing the Workflow Budget-Based Resource Management (WFBB-RM) technique, which can process an open stream of multi-stage jobs with different structures and types of precedence relationships, such as scientific workflows. Lastly, the conclusions of the thesis and directions for future research are presented in Chapter 8.

Chapter 2 Background and Related Work

This chapter presents background information on the concepts, technologies and related work that is relevant to this thesis. In Section 2.1, background information on cloud computing is provided. Section 2.2 then presents a representative set of existing work related to resource management on clouds for processing jobs and workflows with SLAs. The focus of Section 2.3 is on motivating and describing MapReduce, which is a popular multi-stage job that this thesis considers. In Section 2.4, a description of Apache Hadoop, an open-source implementation of the MapReduce programming model, is provided. Resource management techniques for processing MapReduce jobs with various objectives, including MapReduce jobs with deadlines, are presented in Section 2.5. Next, in Section 2.6, a representative set of related work describing techniques to handle error and inaccuracies in user-estimated job execution times is described. Lastly, Section 2.7 compares the techniques described in this thesis with techniques from related work.

2.1 Cloud Computing

In cloud computing, hardware resources (e.g., processor, memory, storage, and network elements) and software resources (e.g., operating systems, tools, and applications) are made accessible on-demand over a network (typically the Internet) [1]. The cloud computing paradigm uses a service-oriented model that offers "everything-as-a-service" (XaaS) [1]. To accomplish this goal, cloud computing utilizes several concepts and technologies, including virtualization, service-orientation, elasticity, scalability, and utility computing. *Virtualization* technology allows the hardware resources (e.g., CPU, memory, storage) of a physical machine to be partitioned into multiple independent *virtual machines* (VMs). A VM is an emulation of a physical machine (e.g., computer) that has its own CPU,

memory, storage, and runs its own operating system. Logically, a VM is viewed as a physical machine but in reality, the VM runs on the physical hardware that may be shared with other VMs. With the help of virtualization technology, cloud computing can deliver an on-demand, service-oriented model that offers: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [1]. IaaS delivers basic computational resources (e.g., virtual machines) as an on-demand service, and PaaS offers a higher-level service (e.g., an application framework with development tools) where consumers can create and deploy their own scalable Web applications without having to invest money to build and maintain their own physical infrastructure. Lastly, SaaS provides consumers with complete end-user applications. Communication and social applications such as Customer Relationship Management systems and email are examples of SaaS.

The *scalability* and *elasticity* characteristics of a cloud provide the ability to grow or shrink the number of resources allocated to a consumer's request dynamically with time. For example, if a Web application that is deployed on the cloud is suddenly exhibiting poor performance due to a surge in traffic leading to a large number of requests, additional resources from the cloud can be deployed to handle the additional requests. Similarly, if the Web application is not using all the resources it has provisioned, resulting in a low resource utilization, money can be saved by scaling down the number of resources that are provisioned. With the *utility computing* model, consumers can lease resources on-demand from the service provider and pay only for the time the resources are used. This is analogous to how people pay for everyday utilities at home such as electricity, gas, and water.

A summary of the primary advantages of the cloud that are attractive to users and has helped expand its popularity is provided next.

- *Low investment and utility computing*: An organization or business can utilize cloud computing for their IT infrastructure instead of purchasing a large number of physical resources and spending money to house/maintain the resources. In addition, the utility computing or "pay-as-you-go" feature of the cloud allows users to acquire resources *on demand* and pay only for the time the resources are used. For example, cloud service providers such as Amazon [8] charge users an hourly rental fee when they provision a virtual machine. This low upfront cost is of great value for small start-up companies as well as larger enterprises who want to reduce costs by migrating some of their business operations to the cloud.
- *Scalability and elasticity*: Cloud computing enables a consumer to dynamically grow and shrink the number of resources provisioned to match the current number of resources required. This provides a significant benefit for handling temporary increases in resource usage and allows consumers to efficiently spend their money.
- *Green computing*: Consumers using cloud computing also contribute to the green computing initiative. By consolidating the IT operations of multiple consumers at a single datacentre instead of each consumer (e.g., organization) maintaining their own IT infrastructure, an effective resource sharing is achieved that can lead to a reduction in power consumed by the computing, storage, and cooling equipment.

There are two main types of clouds that are deployed today: public clouds and private clouds. *Public clouds* offer resources to the general public. Two of today's popular public clouds include: (1) Amazon Elastic Compute Cloud (EC2) [8], which provides IaaS and allows users to deploy virtual machines called *instances*, and (2) Microsoft's Windows Azure [9], which provides both IaaS and PaaS to users. The second type of cloud is a *private cloud* that is only accessible to the members of a given group. For example, two variants of the private cloud are: (1) an *enterprise cloud* that serves the IT needs for employees of a given company, organization, or institution and (2) a *research and engineering* cloud that unifies resources located in multiple institutions to foster resource sharing and collaboration. There is also a third type of cloud called a *hybrid cloud* that borrows features from both private and public clouds. Hybrid clouds provide and manage a set of private resources (owned by an organization, for example), but the hybrid cloud can also provision resources from a public cloud when necessary (e.g., to improve performance or to handle a surge in the number of requests).

2.2 Resource Management on Clouds

Irrespective of the type of cloud deployed, effective resource management strategies and algorithms are needed for harnessing the power of the underlying resource pool of the datacentre. Even though the resource pool of the datacentre is vast, it is still limited and needs to be efficiently shared among multiple users. By using effective resource management strategies, the following benefits can be achieved: higher quality of service for consumers, higher job throughput, lower job response times, and higher utilization of resources [6]. This thesis focuses on the following areas of resource management that is outlined by [6]:

- **Resource allocation (*matchmaking*):** involves determining how to distribute the resources of the cloud economically among competing users and assigning one or more resource(s) to a customer's job.
- **Resource *scheduling***: involves determining when each job assigned to a resource should start to execute.

Matchmaking and scheduling of jobs onto the resources of a cloud are two important operations performed by a resource manager deployed in the resource management middleware of a cloud. When a job arrives, the resource manager invokes a *matchmaking* algorithm that selects the resource(s) from a given pool of resources to be allocated to the job. Once a number of jobs get allocated to a specific resource, a *scheduling* algorithm is used to determine the order in which each of these jobs are to be executed. The matchmaking operation and the scheduling operation are often jointly referred to as the *mapping* operation. Note that in some systems matchmaking and scheduling can be performed in a joint resource management step. The matchmaking and scheduling problem is a NP-hard problem, and the problem becomes even more complex when the user's requirements for quality of service that is often captured in a SLA need to be satisfied, while also achieving the desired system objectives for the service providers, such as maintaining high resource utilization and minimizing the number of overall SLAs that are violated.

2.2.1 Resource Management on Clouds for Processing Jobs with SLAs

In this section, a representative set of work describing techniques for resource management on clouds for processing jobs with SLAs is described. The existing research

is categorised based on the objectives/goals of the research and a discussion of each such category is presented in a subsection.

2.2.1.1 Techniques for Maintaining High Utilization of Resources

In this section, techniques that focus on efficiently provisioning resources in the cloud for achieving high utilization of resources are described.

The authors of [26] design and evaluate a QoS-aware cloud middleware that configures and manages cloud resources based on SLAs. An inefficient approach to guarantee that a job's SLA is not violated is to use a resource overprovision policy, but this is not an optimal solution because it leads to poor system utilization, which in turn can reduce revenue for the service provider. One of the main components of the middleware is a load balancer that monitors the QoS of the system and distributes the load across the platform evenly. In addition, the proposed middleware can dynamically add and remove resources at runtime as needed to meet the SLAs of the jobs.

SmartSLA, a cost-aware resource management framework that addresses how to efficiently manage the resources in a shared cloud database system, is presented in [27]. The objective of the resource management framework is to intelligently allocate the limited resources of a cloud among multiple clients, while ensuring that SLAs are not violated. Machine learning techniques are used to identify the optimal configuration of the system resources (e.g., CPU, memory, and storage) for a client to meet their SLAs, while optimizing the revenue of the cloud service provider.

In [28], the authors focus on the problem of pre-reserved resources in the cloud leading to low resource utilization. Pre-reservation of resources allows users with a complex applications or services to reserve multiple resources ahead of time so that their

SLAs can be satisfied. A resource management mechanism that includes a resource pre-reservation strategy and a resource borrowing/lending strategy is presented. The idea behind resource borrowing/lending is to allow a user to lend idle pre-reserved resources to a user with a shortage of resources. A system controller monitors usage statistics and builds a model that minimizes operating costs while guaranteeing SLAs are not violated.

2.2.1.2 Techniques for Co-allocation and Advance Reservation of Resources

The focus of this section is on techniques for co-allocation and advance reservation of multiple cloud resources, including computing and storage resources.

In [11], the authors discuss the important challenges and architectural elements of SLA-oriented resource management in a cloud environment. Many existing resource management systems in today's datacentres do not yet provide full support for SLA-oriented resource allocation and also do not collectively incorporate customer-driven service management, computational risk management, and autonomic resource management into the resource management system. The authors present an efficient online resource management algorithm that is used in distributed environments for co-allocating resources and supporting advance reservations.

The authors of [29] present an efficient online resource management algorithm that is used in distributed environments for co-allocating resources and supporting advance reservations. Data structures based on 2-dimensional trees are used to organize the temporal availability of resources. The approach uses efficient range searches to identify all the available resources within a specified time window that can be used for co-allocation.

An adaptive resource co-allocation technique for a cloud environment is proposed in [30]. The technique focuses on co-allocating CPU and memory resources and uses a utility function driven approach to optimize resource allocation. More specifically, the authors use a step-wise resource co-allocation approach that repeatedly optimizes the VM placement in each control interval to ensure that load fluctuation can be captured.

The focus of [31] is on the resource provisioning problem for enriched cloud services, which require co-allocating multiple resources in the cloud, including computing, storage, and network resources. Two enriched cloud services are presented: a distributed data storage service and a multicast data transfer service. The authors model and solve the resource provisioning problem for these two services using mixed integer linear programming where the constraints correspond to the QoS requirements of the services.

In [32], the authors focus on devising resource allocation techniques on clouds for applications that require multiple resources (e.g., CPU, network bandwidth, memory and storage). The proposed heuristic based approach includes an on-demand resource allocation mechanism that automatically starts new VMs as required, as well as a load-balancing mechanism to ensure that resources are utilized efficiently.

The authors of [33] focus on the problem of SLA-based resource allocation in a cloud environment for multi-tier applications, where each tier provides a service to the next tier and uses services from the previous tier. A heuristic algorithm based on force-directed search for solving the resource allocation problem is presented. The algorithm optimizes the allocation of processing, memory, and communication resources.

2.2.1.3 *Techniques for Virtual Machine Provisioning and Placement*

In this section, resource management techniques that investigate efficiently provisioning and placing virtual machines (VMs) in a cloud datacentre are described.

In [34], the authors present a resource management framework that comprises two key components: a VM provisioning manager and a VM placement manager. The VM provisioning manager decides how much physical resources (CPU and memory) to allocate to host applications, whereas the VM placement manager decides where to place the application workload within the datacenter. Both the VM provisioning and the VM placement problems are formulated as constraint satisfaction problems and solved using constraint programming where the objective is to ensure that the applications hosted on the cloud meet their SLAs, while also keeping energy costs minimized.

The authors of [35] present an application service provider based resource management technique. The proposed technique formulates the resource management problem as a mixed integer linear optimization problem where the objective is to find the number of VMs that should be allocated to an application to fulfil its SLA, while minimizing the financial cost of provisioning the resources. In addition, the authors also present reactive and proactive heuristic policies to approximate the optimal solution.

2.2.2 *Resource Management on Clouds for Processing Workflows*

Due to cloud computing becoming more prevalent, the various types of applications and workflows executed on clouds have become more extensive. 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 represents the precedence relationships among the tasks.

In [36], a strategy to schedule workflows in a hybrid cloud, which is a private cloud that can request resources from a public cloud on a pay-per-use basis when required, is described. The goal is to determine when and how many resources to request in order to satisfy the deadline of an application while minimizing monetary costs. A Path Clustering Heuristic algorithm is presented to find an initial schedule for the workflow that only considers the resources in the private cloud. If the deadline of the workflow cannot be met, the system decides what type of resources and how many resources to request from a public cloud to ensure that the workflow can meet its deadline.

Scheduling multiple workflows, each one with their own QoS requirements, is the focus of the research presented in [37]. The authors present a scheduling strategy called Multiple QoS Constrained Scheduling Strategy of Multi-Workflows that considers the overall performance of the system (i.e., QoS requirements of multiple users) and not just the completion time of a single workflow.

In [38], a Particle Swarm Optimization (PSO) methodology is used to develop a meta-heuristic based scheduling algorithm to minimize the total monetary cost of executing a workflow application in the cloud. PSO is a stochastic optimization technique that is frequently used in computational intelligence and can be used to solve combinatorial optimization problems. The algorithm attempts to minimize both the execution cost as well as the data transmission (communication) cost of executing workflows in the cloud.

A technique for scheduling workflows on clouds with user-defined QoS requirements such as financial budget constraints and reliability constraints is presented in

[39]. Users submitting the workflow can specify a QoS parameter that they prefer to be optimized, such as make span minimization or monetary cost minimization. Similar to [38], the proposed technique uses a set-based PSO approach where the objective of the technique is to schedule workflows from various users such that as many QoS requirements can be satisfied as possible.

The authors of [40] present a multi-objective optimization framework for workflow task allocation and scheduling on public clouds. The framework uses an extensible cost model and heuristic algorithms to determine the number of virtual machines that should be provisioned to execute a workflow, while also considering QoS parameters such as workflow runtime, communication overhead, and overall financial cost. Both single and multi-objective evolutionary algorithms are used by the framework to perform the resource allocation and scheduling of the workflows.

A heuristic based workflow scheduling algorithm for service-oriented grids, called Partial Critical Paths (PCP), is presented in [41]. The objective of PCP is to minimize the financial cost of provisioning resources for workflow execution, while meeting a user-defined deadline. In [42], the authors adapt the PCP algorithm for a cloud environment and propose two workflow scheduling algorithms based on PCP: a single-phase algorithm called IaaS Cloud Partial Critical Paths (IC-PCP) and a two-phase algorithm called IaaS Cloud Partial Critical Paths with Deadline Distribution (IC-PCPD2). A simulation-based performance evaluation of the algorithms is conducted on a closed system using a batch of synthetic workflows that are based on real scientific applications. The simulation results show that both algorithms are effective, but IC-PCP performs better than IC-PCPD2 in most cases.

2.3 MapReduce

As mentioned in Section 1.2, a popular multi-stage application that this thesis considers is MapReduce. Modern large-scale processing systems should be capable of processing large volumes of data (often referred to as *Big Data*) that are prevalent in today's world. The abundance of data in today's world is a result of the numerous sources of data available, such as:

- scientific data (e.g., health-related data, weather data, and satellite data)
- industrial/organizational data (e.g., financial data, manufacturing data, and retail data)
- business intelligence data (e.g., sales data, customer behaviour data, and product data)
- system data (e.g., system logs, network logs, and status files)

In addition, with the advent of the Internet of Things paradigm leading to a popularity of smart facilities and cyber-physical systems such as sensor-equipped bridges, smart buildings, and industrial machinery [43], a new source of Big Data (from sensors, for example) has emerged. Analyzing this Big Data for making meaningful decisions and obtaining knowledge and insights is important in various types of environments, including enterprise and scientific applications as well as cyber-physical systems.

MapReduce [12] is a programming model, originally proposed by Google, whose purpose is to simplify performing massively distributed parallel processing so that very large and complex data sets can be processed and analyzed efficiently. When dealing with a large volume of data, it is necessary to distribute the computation among multiple

machines to enable parallel processing and reduce the overall processing time. However, there are a number of challenges associated with using parallelism, including how to perform communication and coordination among the different machines, and how to handle and recover from errors and machine failures. In addition, developing and debugging/testing an application that runs on a distributed system is more difficult and complex than developing an application that runs on a single machine. One of the benefits of MapReduce is that it provides an abstraction to hide the complex details and issues of parallelization.

Various companies and institutions use MapReduce, typically in conjunction with cloud computing, for large scale data processing (e.g., sorting, indexing, and grouping) and analyzing very large and complex data sets (i.e., facilitating *Big Data analytics*) [44][45][46]. This includes data mining applications (e.g., web crawling), artificial intelligence applications (e.g., machine learning), and scientific applications (e.g., bioinformatics). For example, Google has previously used MapReduce applications to analyze web documents to generate search indices for its web search engine, and Facebook uses MapReduce to analyze its users' activities and the success of advertisements on its website [47]. Thus, it is common for companies and institutions to submit MapReduce jobs to a private cluster or a cloud for processing. In both cases, effective resource management techniques are required to handle the matchmaking and scheduling of the submitted MapReduce jobs as described in more detail in Section 2.5.

A MapReduce job is characterized by multiple phases of execution, and in the map phase and the reduce phase there are multiple tasks, as illustrated in Figure 2.1. Many computations are expressed using the MapReduce programming model. A classic example

is the *URL access frequency* application that processes the logs of web servers to count the number of distinct URL accesses [12]. This application is a variation of the well-known *WordCount* MapReduce application [12]. The input that needs to be processed is the logs of the Web server. The first step is to split the input data into blocks with a default size of 64 MB, which are called *splits* (refer to Figure 2.1). Next, in the *map* phase, map tasks, which execute a user-defined map function, are created to process each of the splits. The map tasks are independent from each other and can be executed in parallel, possibly on different resources. In the URL access frequency application, the map function reads each URL and generates a set of intermediate key/value pairs in the form: `{URL, 1}`. This key/value pair indicates that one instance of a URL is found. Note that the intermediate dataset generated by the map phase can contain multiple duplicate key/value pairs (e.g., `{www.carleton.ca, 1}` can appear multiple times). In the *shuffle* phase, the intermediary key/value pairs with the same key are grouped together as shown in Figure 2.1 and then passed on to the reduce phase.

During the *reduce* phase, reduce tasks, which execute a user-defined reduce function, process the sorted intermediate key/value pairs to generate the final output, which is typically an aggregate or summary of the original input data that is smaller and more meaningful. Similar to the map tasks, the reduce tasks are independent from one another and can be executed in parallel, possibly on different resources. Note that reduce tasks cannot complete their execution until all the map tasks have finished executing. In the URL access frequency application, the reduce function sums all the values with the same key to emit the output dataset: `{URL, total count}`. Therefore, the final output will be a list of URLs and total number of times each URL is accessed.

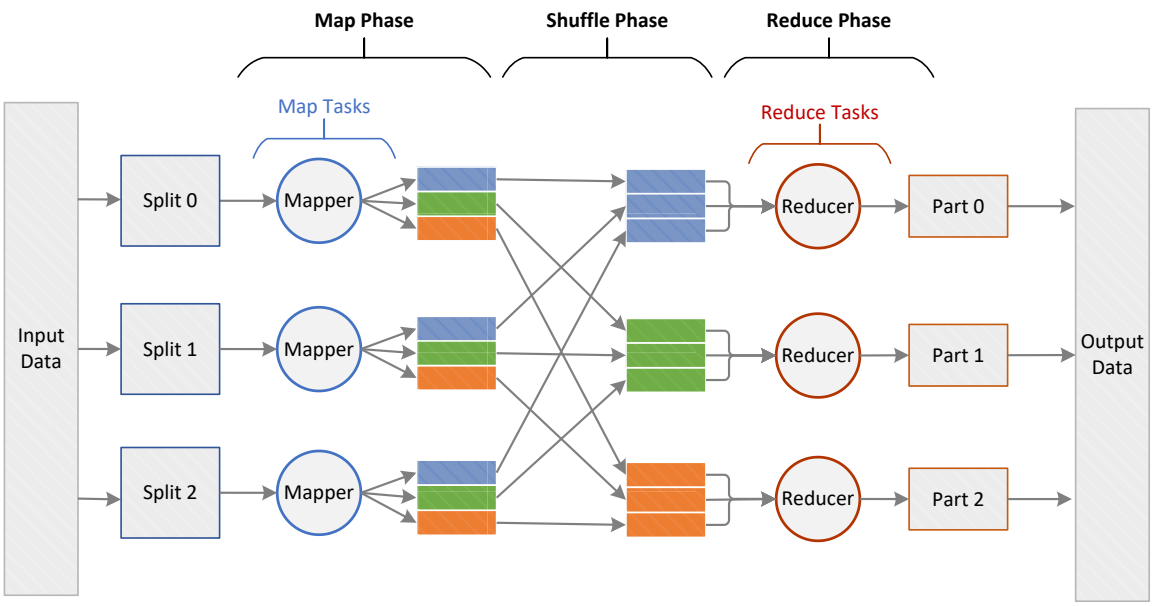

Figure 2.1. Example of a MapReduce job [48].

2.4 Apache Hadoop

Apache Hadoop [25][48] is an open-source software framework (written in Java) that implements the MapReduce programming model (described in the previous section). Hadoop is designed for executing data-intensive distributed computing applications (i.e., Big Data applications), including web analytics applications, scientific applications, applications to perform data mining in social networks, and applications to process and analyze enterprise data [44][49]. The applications that are most suitable for Hadoop are those that require processing data that can be complex, unstructured, and be in a variety of different formats (e.g., XML, JSON, CSV, text, and more) in a parallel (distributed) manner. The Hadoop software framework contains three main sub-frameworks: Hadoop Common, Hadoop Distributed File System (HDFS), and Hadoop MapReduce. *Hadoop Common* provides utility functions including remote procedure call facilities and object serialization libraries that are leveraged by the HDFS and MapReduce frameworks. *HDFS*

is a distributed file system that is based on the Google File System [50], a distributed file system created by Google. HDFS provides redundant storage for the input data required by Hadoop jobs, and it also stores the intermediary data and output data generated by Hadoop jobs. Lastly *Hadoop MapReduce* is an implementation of Google's MapReduce programming model [12].

A set of machines (where each machine is called a *node*) that runs Hadoop is referred to as a *Hadoop cluster* (see Figure 2.2). A typical Hadoop cluster comprises a single *master* node and one or more *slave* nodes. The master node is responsible for maintaining the HDFS and assigning MapReduce tasks to slave nodes for execution. The slave nodes perform work (e.g., read/write to HDFS or execute MapReduce tasks) assigned by the master node. In the original Hadoop MapReduce architecture (MRv1) [51], the master node executes two processes (which are called *Hadoop daemons*): *NameNode* and *JobTracker*. Each slave node also executes two Hadoop daemons: *DataNode* and *TaskTracker*. Note that a discussion on the second version of the Hadoop MapReduce architecture (MRv2) is provided in Section 2.4.1.

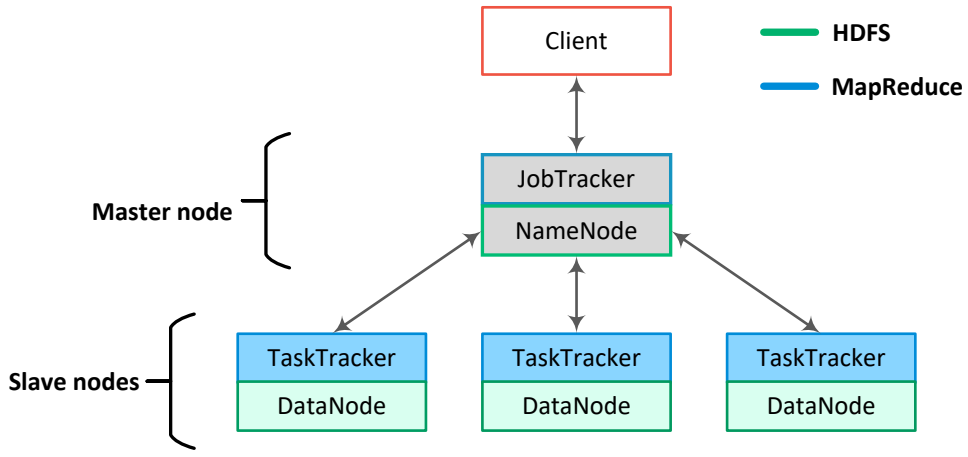

Figure 2.2. Example of a Hadoop cluster using MRv1 [51].

The NameNode and DataNodes are the Hadoop daemons in charge of managing HDFS. Each file that is written to HDFS is split into blocks of 64 MB (default value) and each block is stored on the storage device of the node where a DataNode is running. In addition, each block is replicated three times (default value) and stored on different DataNodes to provide data redundancy and availability. It is the job of NameNode to keep track of which DataNode stores the blocks of each file in the system (which is called the *metadata* of the HDFS). Another important function of NameNode is to direct DataNodes to perform HDFS block operations (creation, deletion, and replication). DataNodes keep in constant contact with NameNode to receive I/O instructions, and they also handle read and write requests from HDFS clients.

An example illustrating how HDFS works is presented in Figure 2.3. Note that in this example, the block replication factor is two. As shown in the illustration, NameNode maintains the metadata of HDFS, and the file named 'file.txt' is composed of two blocks: Block 1A and Block 1B, which are each replicated two times. Block 1A is stored on DataNode 1 and DataNode 2, and Block 1B is stored on DataNode 1 and DataNode 3.

JobTracker provides the connection between the applications that are submitted by users and the Hadoop cluster, and it has the following responsibilities: initialize jobs and prepare them for execution, determine when the map and reduce tasks of jobs should be executed and which TaskTrackers should execute them (i.e., perform matchmaking and scheduling), as well as monitor all the tasks that are currently running. JobTracker is also responsible for managing TaskTrackers, which operate as the JobTracker's slaves and have the primary purpose of executing the map and reduce tasks that JobTracker assigns to them. Each TaskTracker periodically sends polling/update messages (called *heartbeats*) to

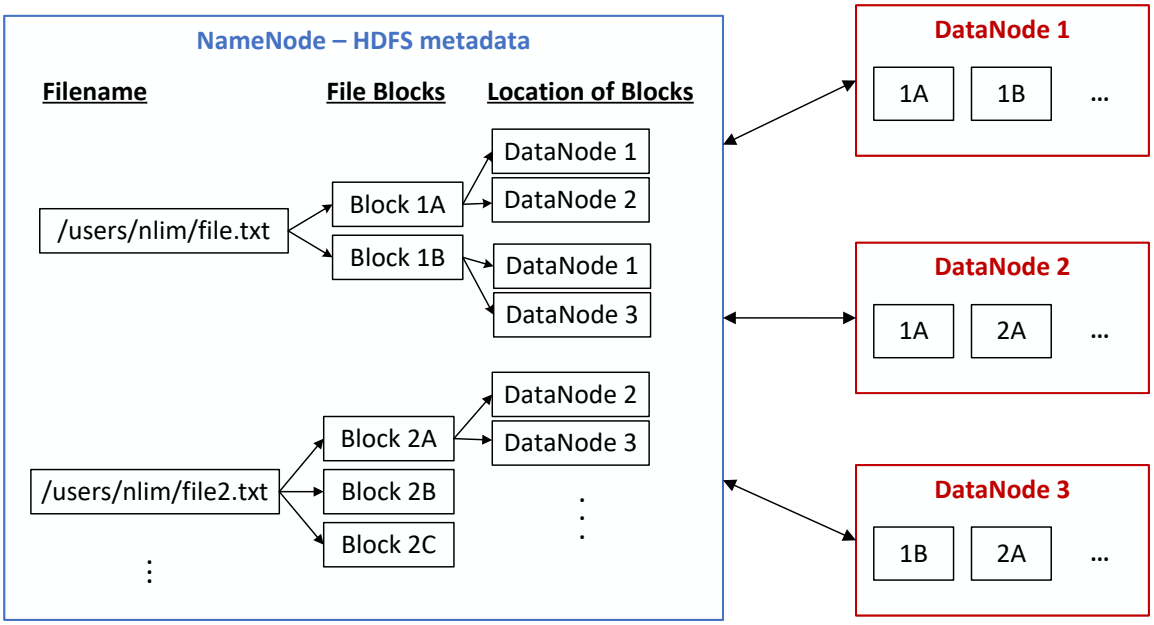

Figure 2.3. Example of HDFS.

JobTracker to update its progress on the tasks it is currently executing (if any) as well as to receive new tasks to execute. If JobTracker does not receive a heartbeat message from a TaskTracker within one minute (default value), JobTracker assumes that the TaskTracker has failed and remaps all the tasks that are assigned to the failed TaskTracker to other available TaskTrackers in the cluster. Each TaskTracker in the Hadoop cluster has a *map task capacity* (or number of *map task slots*) and a *reduce task capacity* (or number of *reduce task slots*), which specify the maximum number of map tasks and maximum number of reduce tasks, respectively, that the TaskTracker can execute in parallel at any point in time.

An illustration of the connection between JobTracker and TaskTrackers is shown in Figure 2.4. As shown in the figure, JobTracker maintains a list of active jobs and completed jobs. There are two active jobs in the example system, and JobTracker has assigned TaskTracker1 two map tasks to execute: Map Task 1 from Job B (Map B1) and Map Task 2 from Job C (Map C2).

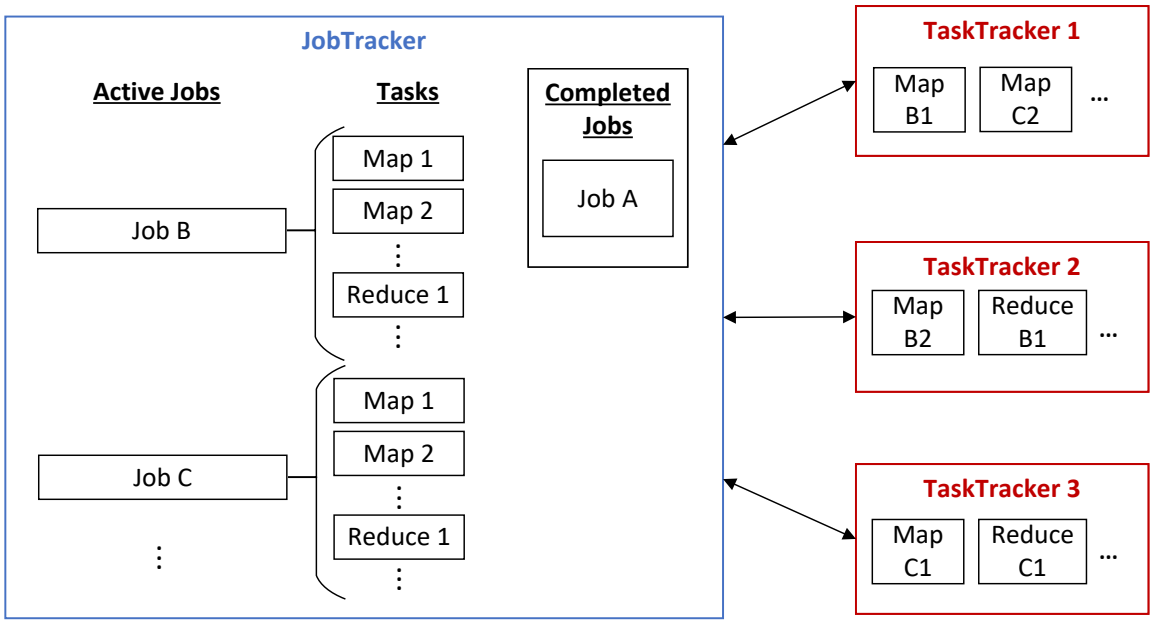

Figure 2.4. Example of Hadoop MapReduce architecture v1.

2.4.1 Hadoop MapReduce Architecture Version 2 (MRv2)

The second version of the Hadoop MapReduce architecture (*MRv2*) is named *Yet Another Resource Negotiator* [52]. The major change in MRv2 from MRv1 is the introduction of a new hierarchical approach that replaces JobTracker and divides its functionality into two main responsibilities: allocation of system resources and job scheduling/monitoring. More specifically, JobTracker and TaskTracker from MRv1 are replaced by three new components (or Hadoop daemons) in MRv2: *ResourceManager*, *NodeManager*, and *ApplicationMaster*. Thus, a Hadoop cluster based on MRv2 has a single ResourceManager daemon running on the master node, a NodeManager daemon executing on each slave node in the cluster, and an ApplicationMaster daemon for each application running on the cluster.

The ResourceManager daemon has two main components: *Scheduler* and *ApplicationManager*. The Scheduler allocates resources (e.g., compute, memory, and

bandwidth) to each of the applications running on the cluster. In MRv2, the resources that an application requires to execute is defined based on the abstract notion of a *resource container*. A resource container defines an application's resource requirements that can include the following: number of CPU cores, memory size, disk size, and network bandwidth. The ApplicationManager is responsible for accepting job submissions from users, obtaining the resource container for starting the ApplicationMaster, and restarting the execution of ApplicationMasters after application or hardware failures. The ApplicationMaster is the Hadoop daemon responsible for negotiating resource containers from the Scheduler for executing the client's application. In addition, ApplicationMasters also work in conjunction with the NodeManagers (of the slave nodes) to execute and monitor the status/progress of the applications, as well as to track and monitor the status and usage of the resource containers. Overall, the changes made in MRv2 improve Hadoop by enhancing reliability and scalability, and enabling greater resource sharing through multi-tenancy.

2.5 Resource Management Techniques for Processing MapReduce Jobs

This section presents a representative set of work related to resource management for MapReduce jobs that have a variety of different objectives, including reducing job completion times, reducing data transmission between resources to minimize network traffic, handling of heterogeneous resources, sharing of resources, and managing the energy consumption of resources. Research with similar objectives are grouped into separate categories and discussed in the respective subsections.

2.5.1 Techniques to Reduce Job Completion Times

The techniques described in this section focus on scheduling MapReduce jobs to minimize job completion times and maintaining high resource utilization.

In [53], the authors present an abstraction of the MapReduce matchmaking and scheduling problem by formulating it as an optimization problem using linear programming where the objective is to find a schedule that minimizes the overall completion time of the jobs in the cluster. Since using linear programming to solve such a problem is NP-hard [54], optimal solutions are difficult and time-consuming to compute even for offline versions on which the details of the batch of jobs to process are known ahead of time. As such, the authors propose algorithms with heuristics to approximate the optimal solutions within a factor of three of the optimal value.

The authors of [55] also model the MapReduce scheduling problem as a linear program where the objective is to minimize the overall completion time of the jobs in the cluster. Two types of jobs are considered. The first type is *data-intensive* jobs which require performing data mining and analysis of very large data sets, including system logs and historical data. The second type are *computationally-intensive* jobs, which are jobs that involve running algorithms or operations with high processing complexity, such as computations involving floating point operations. The modelling of the linear program is based on the traditional job shop scheduling theory. A dispatch-rule based online scheduling policy called LPT-θ that is based on existing algorithms is proposed to approximate the optimal solution.

A MapReduce framework called Dynamically ELastic MapReduce (DELMA) that is capable of dynamically adjusting the cluster size (i.e., adding and removing nodes from

the processing of a job) on the fly is presented in [56]. The main features of DELMA include the following: (1) ability to adjust the cluster size dynamically without having to restart jobs already executing; (2) ability to lower completion time of jobs by adding voluntary or unutilized nodes to the cluster; and (3) ability to replace slow or faulty nodes while a job is being processed.

In [57], a cloud service model for MapReduce named Cura is presented. The objective of Cura is to provide cost-effective MapReduce services in the cloud by implementing an efficient resource allocation scheme that reduces the resource usage cost in the cloud. The core resource management schemes that Cura provides include cost-aware resource provisioning, VM-aware scheduling, and online virtual machine reconfiguration.

2.5.2 Data-Locality-Aware Techniques

MapReduce applications typically process very large datasets and frequent transmission of data from one machine in the cluster to another machine in the cluster over the network can severely reduce system performance due to limited network bandwidth in the cluster. Therefore, it is beneficial to use a *data-locality-aware system* to limit the data transfer between nodes as much as possible. A representative set of data locality-aware techniques are discussed next.

In [58], a scheduling algorithm for workflows comprising multiple MapReduce jobs with precedence relationships is presented. The proposed scheduling algorithm uses a pre-data placement strategy that reduces data transmission over the network, and it also adopts the list scheduling algorithm, which is a priority-based scheduling algorithm. The basic idea of the list scheduling algorithm is to assign each job in the workflow a priority

and schedule the job with the highest priority first. The proposed technique addresses a number of issues including: how to group datasets, where to place them, and how many times to replicate the datasets.

The authors of [59] present a scheduling technique that takes advantage of data locality when scheduling map tasks. The proposed technique attempts to schedule map tasks on nodes that already contain the input data of the respective tasks (referred to as *local map tasks*) in order to prevent time-consuming data transmission over the network. More specifically, the technique gives each node in the cluster a chance to execute any local map task in the queue before non-local map tasks are executed. Experimental results demonstrate that the proposed technique achieves a lower average job response time in comparison to that achieved by a FIFO scheduling technique.

A Locality-Aware Reduce Task Scheduler (LARTS) is presented in [60], which considers data-locality when scheduling reduce tasks. LARTS considers the size and the location of the input data for reduce tasks when making scheduling decisions with the goal of minimizing unnecessary network traffic, which can in turn improve system performance. Through experimentation, the authors showed that using LARTS over the traditional Hadoop FIFO scheduler can lead to a 7% reduction in job execution times.

2.5.3 Techniques for Handling Heterogeneous Computing Environments

This section describes resource management techniques for heterogeneous environments where the resources may have different processing, memory, and network capacities.

In [61], a MapReduce framework called MApReduce with adaptive Load balancing for heterogeneous and Load imbalAnced clusters (MARLA) is presented. MARLA aims

to address the problems that MapReduce implementations, such as Hadoop, have in heterogeneous and load-imbalanced computing environments. The problem with the traditional approach used in other frameworks is that in clusters with heterogeneous resources, nodes that have a lower performance profile may be assigned a similar workload (i.e., equal-sized data partition to process) to those nodes that may exhibit higher performance. MARLA alleviates this problem by using a dynamic task scheduling mechanism that allows each node in the cluster to request tasks at its own pace.

The authors of [62] also focus on resource management for MapReduce workloads in a heterogeneous computing environment. More specifically, a load-balancing algorithm whose purpose is to evenly distribute the workload among nodes with different processing speeds is presented. The algorithm is based on genetic algorithm theory, which is an artificial intelligence-based search heuristic that solves optimization problems by simulating how natural evolution works.

In [63], the authors propose a new approach to solving the MapReduce resource management problem on clouds where the system is characterized by heterogeneous resources. The objective of the proposed approach is to minimize the total financial cost of executing MapReduce jobs on the cloud. The authors model the resource management problem as a constrained combinatorial optimization problem and solve the problem using an innovative constructive algorithm.

2.5.4 Resource Sharing Techniques

Resource management techniques that focus on fairly sharing the resources of a cluster among multiple users as well as techniques that borrow unused resources from other clusters are described in this section.

The *Fair Scheduler* and *Capacity Scheduler*, which are two schedulers that are included with Hadoop, focus on fairly sharing the resources of the cluster among multiple users. The Fair Scheduler [64] is developed by Facebook and its objective is to ensure that each job (on average) gets an equal share of the available resources in the cluster. The idea is to prevent many small jobs from starving the execution of a long job and vice versa. The Fair Scheduler groups jobs into pools and each pool is assigned a minimum share of the cluster's resources (e.g., a minimum number of map task slots and reduce task slots). The Capacity Scheduler [65] is developed by Yahoo and its objective is similar to the Fair Scheduler: share a large cluster among many different independent users (or organizations). Jobs are submitted into queues where each queue is allocated a guaranteed capacity, which is a proportion of the total task slots of the cluster. Note that the unused capacity of a queue can be temporarily allocated to other queues when needed. The jobs within a queue can also be prioritized, where the jobs with a higher priority gain access to the queue's resources first.

A technique called *resource stealing*, which allows currently running tasks to use the unutilized task slots of a node, which the authors refer to as *residual resources*, is presented in [66]. The idea is that when there are available task slots on a node, the system splits the input data of a task into two or more smaller blocks of data and creates an additional sub-task to process each block of data to make use of the unutilized task slots.

In [67], a hierarchical MapReduce framework, which supports executing MapReduce jobs on multiple clusters, such as clusters with unused resources, is described. A hierarchical MapReduce programming model is also proposed where computations are expressed with three functions: Map, Reduce, and GlobalReduce. The input to the

GlobalReduce function comprises the output from all the reduce tasks of a job, and is executed on only one node in the cluster. By supporting the execution of MapReduce jobs in multiple clusters, a more effective resource sharing can be achieved.

2.5.5 Techniques for Energy Management of Resources

This section presents resource management techniques that focus on green computing issues in the context of MapReduce jobs: minimizing the energy consumed by a distributed system, such as a cloud or cluster, when executing MapReduce jobs.

The authors of [68] investigate techniques to improve the energy-efficiency of running MapReduce jobs in datacentres and computational grids without severely affecting performance. The authors study the performance and energy-efficiency trade-offs of Hadoop using various workloads. The system activity traces that were recorded during experiments show that MapReduce computations involve a large number of I/O operations (e.g., reading/writing a large volume of data from/to disks), as well as network I/O operations) leading to low CPU utilization at various points in time. Through their study, the authors have found that careful resource allocation to match an application's degree of parallelism and using the well-known dynamic voltage and frequency scaling technique can improve energy-efficiency without a large performance cost.

The focus of [69] is also on the challenge of making the execution of MapReduce jobs more energy-efficient. The authors consider a very bursty MapReduce workload with distinct CPU, memory, and network requirements that is executed on a heterogeneous datacentre. An online energy minimization path algorithm called Green MapReduce Scheduler (GEMS) for scheduling MapReduce jobs is presented. GEMS reduces energy

consumption while maintaining a low task response time by using sleeping policies on the compute servers and the network switches simultaneously.

2.5.6 Techniques for Handling MapReduce Jobs with Deadlines

This section presents a representative set of work describing techniques for processing MapReduce jobs with deadlines. MapReduce jobs with an associated deadline for completion have recently become important for latency-sensitive applications [70] such as those used in the context of live business intelligence, personalized advertising, spam/fraud detection, real-time analysis of event logs, and various additional real-time data analytics applications. Business intelligence refers to analyzing the raw data of a business or corporation so that effective business strategies can be developed and more informed business decisions can be made. Event log analysis involves processing event logs to find specific patterns, filter event occurrences, and group similar event occurrences together. Such event log analysis can be used for various types of computing systems that have event monitors to collect and signal event occurrences, including operating systems, database management systems, and cyber-physical systems. More generally, allowing users to specify deadlines, allows the system to prioritize jobs and ensure that time-critical jobs are completed on time. In some situations, it is ideal to analyze the most up-to-date data and receive the results in a timely manner so that the best decisions can be made. Thus, it is common for various companies and institutions to submit MapReduce jobs with deadlines to a cluster or a cloud for processing. An important component of this thesis is on devising effective and efficient resource management techniques for processing an open stream of MapReduce jobs with SLAs, where each SLA is characterized by an earliest start time, an execution time, and a deadline.

The authors of [71] propose a Deadline Constraint Scheduler for Hadoop to process jobs with deadlines. A job execution cost model is devised that considers parameters such as the execution time of map tasks, the execution time of reduce tasks, and the size of the input data to process. This model is used to perform a schedulability test to determine if a submitted job can be completed before its deadline given the current available resources in the cluster. If the job cannot meet its deadlines, users have the option of changing the deadline requirements and resubmitting the job.

In [72], the authors investigate the problem of scheduling MapReduce workloads comprising jobs with deadlines as well as jobs without deadlines. The authors present a scheduler that adopts a sampling-based technique called Tasks Forward Scheduling (TFS) to predict the execution times of map tasks and reduce tasks. TFS predicts the execution times of tasks by initially executing a few tasks and then using the actual runtimes of these initial tasks to predict the execution times of future tasks. In addition, the proposed scheduler also leverages a resource allocation model named Approximately Uniform Minimum Degree of Parallelism to dynamically control the execution of each job such that the job executes at its minimum degree of task parallelism to meet its deadline. The idea is to prevent a single job from monopolizing all the resources in the cluster and to allow more jobs to be executed on the cluster in parallel.

In [70] two resource allocation policies based on earliest deadline first (EDF) are presented. The first policy is called Minimum Resource Quota Earliest Deadline First (MinEDF), which allocates the minimum number of task slots required for completing a job before its deadline (similar to [72] described earlier). The second policy is called Minimum Resource Quota Earliest Deadline First with Work-Conserving Scheduling

(MinEDF-WC). MinEDF-WC enhances MinEDF by adding the ability to dynamically allocate and deallocate resources (task slots) from active jobs when required. This ability to dynamically allocate and deallocate resources allows a machine with spare resources to share its unused resources with other jobs that need them.

A policy for dynamic provisioning of public cloud resources to schedule MapReduce jobs with deadlines is described in [73]. 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 present a resource provisioning policy that aims to minimize the number of resources that are provisioned from the cloud since provisioning resources from the cloud incurs a financial cost.

The authors of [74] investigate resource management algorithms for minimizing the cost of allocating virtual machines to execute MapReduce jobs with deadlines. Two VM provisioning strategies are proposed: (1) List and First-Fit (LFF) and (2) Deadline-aware Tasks Packing (DTP). The LFF approach sorts the pricing policies of VMs according to either increasing order of unit cost or decreasing order of VM performance. Each map task is assigned to its own VM and reduce tasks are assigned to one of the VMs already provisioned for map tasks. In the DTP approach, 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 [75], 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. Moreover, a new MapReduce scheduler for processing MapReduce

jobs with deadlines based on bipartite graph modelling called the Bipartie Graph Modeling MapReduce Scheduler (BGMRS) is presented in [76]. 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 by transforming the problem into a well-known graph problem: minimum weighted bipartite matching.

2.6 Handling Error/Inaccuracies in User-estimated Job Execution Times

Estimates of job runtimes provided by users are often error prone/inaccurate and users tend to overestimate the runtimes of their jobs [16][17][18]. The error/inaccuracy in user-estimated execution times can be detrimental to system performance. This is because a matchmaking and scheduling algorithm makes decisions based on a user's estimated execution times, and thus errors/inaccuracies can diminish the quality of the resource management decisions that are made [77][78]. A representative set of work describing techniques for handling error associated with user estimates of job execution times is presented next.

In [77], a middleware framework for grids that provides robustness by handling error/inaccuracies in user-estimated job execution times is presented. The authors describe a pre-scheduling mechanism based on overbooking, which can prevent unnecessary rejection of jobs when the user-estimated execution times are overestimated. Overbooking allows a small proportion of jobs to miss their deadlines, which means that even if the schedulability analysis determines a job cannot meet its deadline, the job can still be accepted as long as the proportion of late jobs remains lower than the overbooking threshold. Note that other works (see [79] and [80], for example) have also used a similar

overbooking mechanism to handle jobs with overestimated execution times. The authors also present a Schedule Exceptions Manager that monitors the resource schedule and adapts the resource schedule when there are overestimated/underestimated job execution times.

The authors of [81] present a Soft Advance Reservation (SAR) technique for grids subjected to advance reservation requests. The SAR technique relaxes the requirement that all advance reservation requests, which are typically characterized by an earliest start time, an execution time, and a deadline for completion, must meet their deadlines. In addition, a technique for handling errors associated with user-estimates of job runtimes is also presented. The technique described adopts a history-based approach where the previous two requests submitted by the same user is used to compute a system-generated estimated runtime for the current request.

In [82], the authors present a technique to handle error in user-estimated job execution times for a popular scheduling algorithm, named First-come-first-serve (FCFS) with backfilling, used in parallel supercomputing environments. First, the algorithm calculates a system-generated prediction of the job runtime and uses this value for scheduling. The system-generated predicted runtime is calculated using the most recent historical data (e.g., average runtime of the two previous submitted jobs from the user). If the system's prediction of the job runtime is too short, it is extended, and the job can run until it finishes or until it reaches the user-estimated runtime, which serves as the kill-time of the job. Using the historical data of the job runtimes at various degrees has also been used by other researchers (see [83], [84], and [85], for example) to predict the runtime of future requests.

The research presented in [86] also concerns improving scheduling algorithms based on backfilling, specifically focusing on how to handle jobs with underestimated execution times. Similar to [82], the improved algorithm does not abort jobs with underestimated runtimes. Instead, the jobs can run for an extended period if it does not cause other jobs to be delayed. If extending the job's runtime does affect other requests in the system, the job is aborted. Experimental results show that the improved backfilling algorithm reduces the number of aborted requests significantly.

The research in [78] also investigates the effect of job execution time estimation error on algorithms that depend on user-estimated job runtimes such as backfilling algorithms and the shortest job first algorithm. The authors present techniques for adjusting the user-estimated runtimes based on historical data of the accuracy of previous user estimates of job runtimes. More specifically, the user-estimation accuracy (denoted R) is defined as the ratio of the request's actual runtime over the user estimation of the request runtime. The objective is to generate more accurate job runtimes to allow the scheduler to make more intelligent scheduling decisions for improving system performance. A number of different schemes are presented for determining which kind of historical data to use to calculate R, including a user-based scheme that uses historical data from the same user, a project-based scheme that uses the historical data from the same project, and a combined scheme that uses both these types of historical data.

2.7 Comparison of Thesis Research with Related Work

A wide variety of issues regarding resource management on clouds for processing jobs with SLAs are described in the literature and were reviewed in the previous sections. Most of the work in the literature has only addressed meeting deadlines for jobs that require

a single resource or handling a batch workload that comprises a fixed number of jobs executing on the system. To the best of our knowledge, none of the related works focus on the problem of meeting an end-to-end SLA (characterized by an earliest start time, an execution time, and an end-to-end deadline) for multi-stage jobs that require service from *multiple* resources, such as workflows and MapReduce jobs, on an open system subjected to a stream of job arrivals. This is the focus of attention for this thesis.

With respect to MapReduce jobs, none of the existing works have dealt with all the aspects of the problem that this thesis concerns. The works described in Section 2.5.1 to Section 2.5.5 do not consider jobs with deadlines, but instead focus on other aspects of matchmaking and scheduling of MapReduce jobs. Moreover, the works described in Section 2.5.6, which do consider MapReduce jobs with deadlines, do not consider jobs with SLAs characterized by an earliest start time, which is important in the context of advance reservation requests. Furthermore, the techniques described in Section 2.5.6 (except [70]) do not handle workloads comprising an open stream of job arrivals, instead a batch workload comprising a fixed number of jobs is used. Note that the resource management techniques described in this thesis are compared with the techniques described in [70] (see Section 4.5). Additionally, the default schedulers that Hadoop [25] come installed with, including the FIFO, Capacity, and Fair schedulers, do not support jobs with deadlines. FIFO simply schedules jobs in a first-in first-out manner whereas the Capacity and Fair schedulers (which are described in Section 2.5.4) focus on fairly sharing the cluster's resources among multiple users. Other third party Hadoop schedulers, such as the adaptive scheduler [87], dynamically alter the number of resources when required, whereas this thesis concerns systems with a fixed number of resources.

A key contribution of this thesis is presenting techniques to handle error/inaccuracies in user-estimated job execution times that are submitted as part of the SLA of the job. In particular, techniques are presented for processing MapReduce jobs with inaccurate job execution times. None of the techniques described in Section 2.5.6, which consider MapReduce jobs with deadlines, handle errors/inaccuracies with user-estimates of job runtimes. On the other hand, the techniques described in Section 2.6 do consider the handling of errors associated with user-estimated job runtimes but these techniques do not deal with all the aspects of the problem that this research focuses on: matchmaking and scheduling an open stream of MapReduce jobs with SLAs. Most of the work on techniques for handling errors/inaccuracies in user-estimated job runtimes focus on scheduling algorithms used in parallel supercomputing environments, including first-come-first-serve with backfilling, and do not concern resource management for environments that process MapReduce applications. Moreover, there is a wide body of research that describes techniques for predicting the runtimes of the jobs such as those described in [83][84][85]. However, previous investigations have shown that the system predicted runtimes can still be error prone [77]. The techniques described in this thesis can tolerate errors/inaccuracies in user-estimated job runtimes, whether provided by the user or generated by the system, to avoid degradation in system performance.

In summary, this thesis focuses on investigating issues in resource management on clouds that have not been addressed in the current state of the art: handling workloads characterized by an open stream of multi-stage jobs with SLAs and effectively handling errors associated with user-specified execution times in such workloads.

Chapter 3 Resource Management Techniques for Processing a Batch of MapReduce Jobs with SLAs

In this chapter, resource management techniques for processing a batch of MapReduce jobs with SLAs are described. More specifically, the resource management techniques that are presented formulate and solve the matchmaking and scheduling problem as an optimization problem. The rest of the chapter is organized as follows. First, the problem description and model is presented in Section 3.1. Next, Section 3.2 provides an overview of the approach for solving the resource management problem. The focus of Section 3.3 and Section 3.4 are on describing the formulation of the resource management problem using optimization techniques. In Section 3.5, the experiences in implementing the optimization models using various software packages are discussed. Section 3.6 and Section 3.7 present and discuss the performance evaluation of the resource management techniques. Lastly, a summary and discussion of the chapter is provided in Section 3.8.

3.1 Problem Description and Model

This section provides a model for matchmaking and scheduling MapReduce jobs with SLAs comprising an earliest start time, an execution time, and an end-to-end deadline. The workload comprises a set (batch) of MapReduce jobs to execute, $J = \{job\ 1, job\ 2, \ldots, job\ n\}$ where n is the number of jobs in the set. Each job j (j = 1, 2, …, n) in the set J has the following attributes:

- A set of map tasks: $T_j^{mp} = \{\tau_{j,1}, \tau_{j,2}, \ldots, \tau_{j,k_j^{mp}}\}$ where k_j^{mp} denotes the number of map tasks in job j.

- A set of reduce tasks: $T_j^{rd} = \{\tau_{j,1+k_j^{mp}}, \tau_{j,2+k_j^{mp}}, \ldots, \tau_{j,k_j^{rd}+k_j^{mp}}\}$ where k_j^{rd} denotes the number of reduce tasks in job *j*.
- An earliest start time (or release time) (s_j), which is the earliest time at which job *j* can start to execute.
- Deadline (d_j) by which the job should be completed (i.e., soft deadline). A job with a soft deadline is permitted to miss its deadline; however, the desired system objective is to minimize the number of jobs that do miss their deadlines.

A set called *AT* is defined as the union of T_j^{mp} and T_j^{rd} for all the jobs *j* in *J* (*j* = 1, 2, …, *n*): $AT = \bigcup_{j \in J} (T_j^{mp} \cup T_j^{rd})$. Thus, *AT* contains all the tasks of all the jobs. Each task *t* ($t \in AT$) is either a map task or a reduce task and is characterized by an execution time (in seconds), e_t, and a resource capacity requirement (q_t) that specifies the number of resources a task needs to execute. A typical map and reduce task only requires executing on one resource [48], and therefore q_t is set to 1. The execution times of the map tasks includes the time required to read the input data, and the execution times of the reduce tasks includes the time required to exchange data (e.g., intermediate keys) between the map phase and reduce phase.

The set of MapReduce jobs, *J*, is executed on a distributed/parallel computing environment, which is represented by a set of resources, $R = \{res\,1, res\,2, \ldots, res\,m\}$ where *m* is the number of resources in the system. Such an environment can represent a private cluster, or a set of nodes acquired a priori from a cloud (e.g., Amazon EC2) for processing the MapReduce jobs arriving on the system. Each resource *r* in *R* (*r* =1, 2, ..., *m*) is modelled after a Hadoop TaskTracker (described in Section 2.4) where each resource

has a map task capacity (number of *map task slots*), c_r^{mp}, and a reduce task capacity (number of *reduce task slots*), c_r^{rd}. The map task and reduce task capacities specify the maximum number of map tasks and reduce tasks, respectively, that each of the resources can execute in parallel at any point in time.

The requirements for matchmaking and scheduling the set of jobs J on to the set of resources R are summarized next. Each task t in AT can only be scheduled to start at or after job j's earliest start time, s_j. Secondly, each task t in AT can only be mapped to a single resource r where t executes on r for e_t time units. For a job j, all the map tasks of job j must complete executing before the reduce tasks of job j can start executing. Furthermore, at each point in time, the capacity limits of the resources cannot be violated (i.e., a resource cannot be assigned to run more tasks in parallel than its capacity). The objective of the system is to minimize the number of jobs that miss their deadlines. Note that the *laxity* of a job can be used to determine how stringent the job's deadline is, and it can be used to help a system prioritize which jobs to execute first. A discussion on the laxity of jobs is provided in the upcoming sub-section.

3.1.1 Laxity of Jobs

The *laxity* (also called *slack time*) of a job is the extra time that a job has for meeting its deadline if it starts executing at its earliest start time, and it is used as an indicator for how stringent the deadline of a job is. The laxity of a job j (denoted L_j) is calculated as follows:

$$L_j = d_j - s_j - SET_j \tag{3.1}$$

where d_j is the deadline of job j, s_j is the earliest start time of job j, and SET_j is the *sample execution time* of job j. SET_j is calculated with the user-specified task execution times of

the job. More specifically, SET_j can be calculated in one of three ways: (1) *maximum job execution time*—assumes the job is executed on a system with a single resource (denoted SET_j^{max}), (2) *minimum job execution time*—assumes the job is executed at its maximum degree of parallelism (denoted SET_j^{min}), or (3) execution of the job when it executes at is maximum degree of parallelism on a set of resources R with m resources (denoted SET_j^R). SET_j^R is determined by mapping job j on to a set of resources that represent the resources in R, assuming that job j is the only job in the system. The tasks of job j are mapped by using the "*ready* tasks with the highest execution time first". Tasks are considered *ready* when all of their preceding tasks have completed executing and are mapped in non-increasing order of their execution times. Each task is scheduled to start executing at its earliest possible time on the m resources.

3.2 Overview of the Approach

The matchmaking and scheduling problem (described in the previous section) is formulated and solved as an optimization problem using mixed integer linear programming (MILP) [13] and constraint programming (CP) [14]. Both MILP and CP are well-known theoretical techniques that can solve optimization problems and have been shown to be effective in solving planning and scheduling problems, such as the traditional job shop scheduling problem [88]. Thus, the use of MILP and CP lead to an optimal solution in the sense that the schedule that is generated results in the number of jobs that miss their deadlines being minimized. Both MILP and CP have the same general modelling structure: decision variables, objective function, and constraints. The *decision variables* are initially unknown and are assigned values once the problem is solved (i.e., they are the output of the model). The *objective function* is a mathematical function that generates the value that

needs to be optimized (minimized or maximized). Lastly, the *constraints* are a set of mathematical formulas that restrict the values that the decision variables can be assigned. Solving the optimization model involves assigning values to the decision variables to optimize the value generated by the objective function, while ensuring that none of the constraints are violated.

MILP is a subfield of mathematical programming (also referred to as mathematical optimization) where the model has the following characteristics: (1) some of the decision variables must be integers and (2) the objective function and constraints are mathematically linear [13]. The theoretical basis for MILP and mathematical optimization in general is numerical algebra [89]. To solve MILP problems, techniques such as cutting-planes (constraint relaxations) and Branch and Bound are used. The theoretical foundation for CP is different than that of MILP. CP was developed by computer science researchers in the mid-1980s by combining knowledge and techniques from artificial intelligence, logic and graph theory, and computer programming languages [14]. Search algorithms, including back-tracking and local search [14], are commonly used to solve CP models. The general idea in these search algorithms is to use logical inferences to assign values to the decision variables and then evaluate if the new values of the decision variables produce a better output (higher value if maximizing or lower value if minimizing) for the objective function.

Unlike MILP models, CP models natively support a variety of arithmetic operators and logical constraints such as integer division and the 'implies' constraint [90]. To formulate logical constraints in a MILP model, the 'big-M' formulation technique [13] is typically used. In addition, CP also defines a general set of specialized constraints, called *global constraints*, that model frequently used patterns seen in optimization problems [91].

For example, one such constraint is the *cumulative* constraint, which is often used in scheduling problems to ensure that the capacity of each resource is not violated at any point in time. One of the limitations of CP models is that, natively, the decision variables can only be discrete (i.e., integer or Boolean) [14], whereas MILP models can support both discrete and continuous decision variables.

Figure 3.1 provides an illustration of the approaches that are used to solve the matchmaking and scheduling problem. As described in Section 3.1, the input required by the resource management model consists of a set of jobs J and a set of resources R. The matchmaking and scheduling problem that is formulated using CP is called the *CP Model* and is discussed in detail in Section 3.3. Similarly, the matchmaking and scheduling problem that is formulated using MILP is called the *MILP Model* and is described in Section 3.4. Three implementations of the MILP Model and CP Model using different software packages are considered:

- *Approach 1*: the MILP Model is implemented and solved using LINGO [92] (commercial software).
- *Approach 2*: the CP Model is implemented using MiniZinc/FlatZinc [93] and solved using Gecode [94] (both open source software).
- *Approach 3*: the CP Model is implemented and solved using IBM ILOG CPLEX Optimization Studio (abbreviated CPLEX) [15] (commercial software).

The *output* produced after solving the resource management model includes the following: (1) the assigned resource and scheduled start time for the tasks of each job, (2) the completion time of the batch of jobs, and (3) the number of jobs that miss their deadlines. The *measurements* that are made on the system to evaluate the different

approaches is the processing time required by the respective CP or MILP solver to produce the output. In general, the complexity of solving a MILP or CP problem is NP-Complete but state-of-the art solvers can make optimizations to solve problems in polynomial time [14]. Since commercial solvers, such as IBM CPLEX [15], are proprietary and their algorithms are often unknown, it is difficult to theoretically define the complexity of solving the problem. Thus, the complexity of the algorithms is evaluated empirically using experimentation as described in Section 3.6.

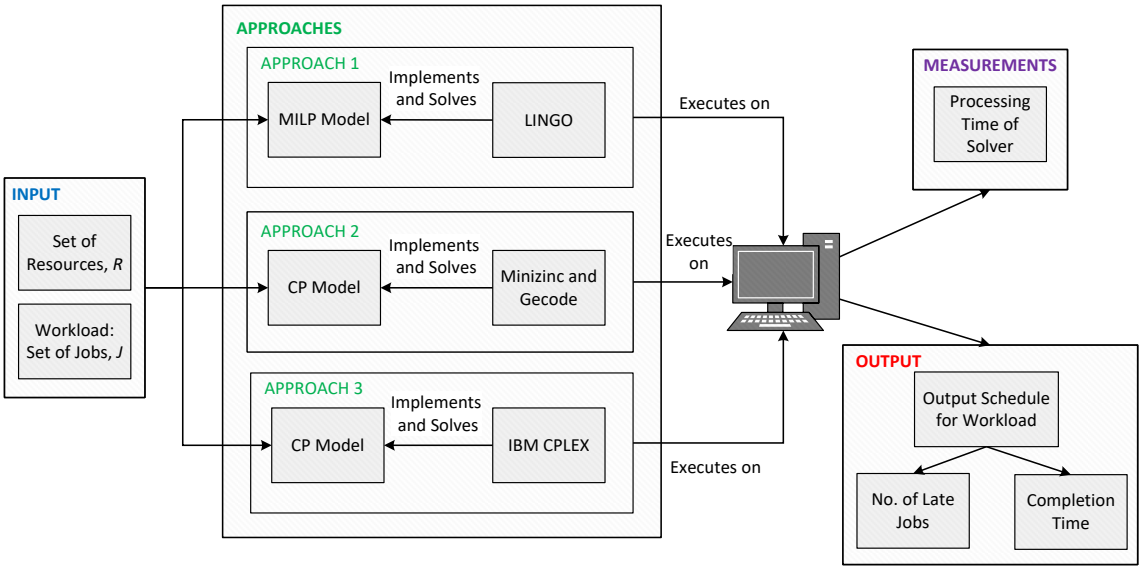

Figure 3.1. Overview of approaches for solving the resource management problem using optimization techniques.

This chapter of the thesis is motivated by the following issues. The first is how to employ the existing theory on MILP and CP for devising efficient resource management algorithms that can minimize the number of jobs missing their deadlines on a closed system subjected to a batch workload comprising MapReduce jobs with deadlines. The second issue is the development of efficient implementations of the algorithms using commercial-

off-the-shelf and open source software packages that produce an acceptable system overhead accrued during the execution of the resource management algorithms.

3.3 Formulation of the CP Model

The formulation of the CP Model is presented in Table 3.1. Recall from Section 3.1 that the input required by the CP Model comprises a set of resources R and a set of jobs J. In addition, a set AT contains all the tasks of all the jobs in J. The decision variables of the CP Model are outlined:

- x_{tr}: A binary variable for *matchmaking*, which is set to 1 if a task t is assigned to a resource r; otherwise, x_{tr} is set to 0. There is an x_{tr} variable for all combinations of task t in AT and resource r in R. Thus, the number of x_{tr} decision variables is equal to the product of the number of tasks in AT and the number of resources in R.
- a_t: An integer variable for *scheduling*, which specifies the assigned (or scheduled) start time of a task t. There is an a_t variable for each task t in AT.
- N_j: A binary variable that is set to 1 if job j misses its deadline. Each job j in J has an N_j variable that is initialized to 0.

Note that an integer variable is used for a_t because CP does not support real values as discussed in the previous section. Although time is a continuous variable, discrete time values can be considered by changing the unit of time. For example, if the execution of a task takes 3.8 minutes, the time can be converted into an integer value by changing the unit of time to seconds (i.e., 228 seconds). In some cases, if the lengths of times are very different (e.g., 0.8 sec versus 1000 sec), it may not be ideal to change the unit of time because the converted values can be quite large: 0.8 sec becomes 800 ms and 1000 sec becomes 100000 ms. In this case, it may be more appropriate to round the non-discrete

time values to the nearest higher integer. For instance, the 0.8 sec can be rounded up to 1 sec.

Table 3.1. CP Model.

$$Minimize \sum_{j \in J} N_j \qquad such\ that$$

$\sum_{r \in R} x_{tr} = 1 \quad \forall\, t \in AT$	(1a)
$\left(a_{t^{mp}} \geq s_j \quad \forall t^{mp} \in T_j^{mp}\right) \quad \forall\, j \in J$	(2a)
$\left(a_{t^{rd}} \geq \max_{t^{mp} \in T_j^{mp}} \left(a_{t^{mp}} + e_{t^{mp}}\right) \quad \forall t^{rd} \in T_j^{rd}\right) \forall j \in J$	(3a)
$\left(\max_{t^{rd} \in T_j^{rd}} \left(a_{t^{rd}} + e_{t^{rd}}\right) > d_j \implies N_j = 1\right) \quad \forall j \in J$	(4a)
$\left(cumulative\ (\{a_t \mid x_{tr} = 1\}\ \forall t \in T^{mp}, \{e_t \mid x_{tr} = 1\}\ \forall t \in T^{mp}, \{q_t \mid x_{tr} = 1\}\ \forall t \in T^{mp}, c_r^{mp})\right) \forall r \in R$ where $T^{mp} = \bigcup_{j \in J} \left(T_j^{mp}\right)$	(5a)
$\left(cumulative\ (\{a_t \mid x_{tr} = 1\}\ \forall t \in T^{rd}, \{e_t \mid x_{tr} = 1\}\ \forall t \in T^{rd}, \{q_t \mid x_{tr} = 1\}\ \forall t \in T^{rd}, c_r^{rd})\right) \forall r \in R$ where $T^{rd} = \bigcup_{j \in J} \left(T_j^{rd}\right)$	(6a)
$(x_{tr} \in \{0, 1\} \quad \forall\, t \in AT) \quad \forall r \in R$	(7a)
$N_j \in \{0, 1\} \quad \forall j \in J$	(8a)
$a_t \in \mathbb{Z} \quad \forall\, t \in AT$	(9a)

Table 3.1 shows that the objective function of the CP Model states that the number of late jobs should be minimized. This is accomplished by minimizing the sum of all the N_j variables. Recall that N_j is set to 1 if a job j misses its deadline. A description of the constraints of the CP Model is provided next. Constraint (1a) states that each task t in AT can only be assigned to a single resource. This is accomplished by summing all the x_{tr}

variables of a given task t and asserting that the sum is equal to 1. Ensuring that the map tasks for each job j in J (stored in set T_j^{mp}) has an assigned (or scheduled) start time (a_t) that is at or after the job j's earliest start time (s_j) is captured by constraint (2a). Constraint (3a) enforces that the reduce tasks for each job j in J (stored in set T_j^{rd}) are scheduled to start after all the map tasks of the respective job j are finished executing. The time at which all the map tasks finish executing is equal to the completion time of the *latest finishing map task* (LFMT). To find the LFMT, constraint (3a) iterates through all the map tasks of a given job j (stored in T_j^{mp}) and uses the *max* function to find the task with the highest completion time, where the completion time of a task t is equal to the sum of a_t and t's execution time (e_t). Note that the *max* function returns the maximum value from a given set of values.

Ensuring that the N_j for all the jobs that miss their deadlines is set to 1 is the purpose of constraint (4a). A job j misses its deadline (d_j) if the completion time of the *latest finishing reduce task* (LFRT) exceeds d_j. The completion time of the LFRT is calculated in a similar manner to how the completion time of the LFMT is calculated. The difference is that the job j's reduce tasks (stored in T_j^{rd}) are passed to the *max* function instead of job j's map tasks. Note that the CP Model makes use of the logical operator, *implies* (=>) in constraint (4a). The next two constraints (5a) and (6a) are the resource capacity constraints and impose that the map and reduce task capacities of each resource in R are not violated at any point in time. These constraints are formulated using the CP global constraint function, *cumulative* [91]. For each point in time, the cumulative function sums up the number of executing tasks at the given time point and ensures that this number does not exceed the resource capacity limit. Four parameters are required by the cumulative

constraint: a set of scheduled start times of the tasks, a set of execution times of the tasks, a set of the resource requirements of the tasks, and the capacity of the resource. There are two cumulative constraints for each resource r in R: one for resource r's map task slots and one for resource r's reduce task slots. For a particular constraint, only the tasks that are assigned to that resource (i.e., $x_{tr} = 1$) are included as input for the constraint. Lastly, the remaining constraints (7a) to (9a) define the domain of the decision variables, which are the valid values that the decision variables can be assigned.

Complexity of the CP Model: The number of decision variables and the number of constraints in the CP Model are directly proportional to the number of jobs, number of tasks, and number of resources in the input sets *J, AT*, and *R*, respectively. The number of decision variables was discussed earlier and the number of constraints is discussed next. For each task t in AT, there is a constraint (1a) and a constraint (9a). Furthermore, for each map task of each job in J, there is a constraint (2a), and for each reduce task of each job in J, there is a constraint (3a). There is also a constraint (4a) and a constraint (8a) for each job in J. Moreover, for each resource in R, there is a constraint (5a) and a constraint (6a). Lastly, the number of constraint (7a) in the CP Model is equal to the product of the number of resources in R and the number of tasks in AT. In the experiments performed (refer to Section 3.7), the CP Model was observed to give rise to a reasonable memory requirement and CPU time requirement.

3.4 Formulation of the MILP Model

The MILP Model is presented in Table 3.2, and it uses a time-indexed formulation [95], which is a commonly used model for formulating scheduling problems using discrete time (i.e., integer values for time). Recall from the previous section that although time is a

continuous variable, discrete time values can be used by changing the unit of time or by rounding the values. The discrete time values are contained in a set I called the *time range*. Similar to the CP Model, the input required by the MILP Model is a set of resources (R) on which to execute the set of jobs (J), and there is a set AT that contains all the tasks of all the jobs in J. The MILP Model uses the following decision variables:

- x_{tri}: A binary *matchmaking* and *scheduling* variable that is 1 if a task t is assigned to start executing on a resource r at time i; otherwise, x_{tri} is 0. There is an x_{tri} variable for each combination of tasks t in AT, resource r in R, and times i in I.
- N_j: A binary variable that denotes if a job misses its deadline. N_j is set to 1 if job j misses its deadline; otherwise N_j is set to 0. There is an N_j variable for each job j in J, and N_j is initially set to 0 for all the jobs.

The objective function of the MILP Model is identical to that of the CP Model: minimize the number of jobs that miss their deadlines. Furthermore, the constraints of the MILP Model have the same functionality as that of the CP Model (shown in Table 3.1); however, the MILP Model's constraints are expressed differently. This is because the MILP Model uses a single decision variable for matchmaking and scheduling (x_{tri}), whereas the CP Model defines a separate decision variable for matchmaking (x_{tr}) and for scheduling (a_t). In addition, the CP Model leverages CP's global constraints and native support for mathematical operators such as the 'implies' operator ($\Longrightarrow$). A walkthrough of the MILP Model's constraints is provided next.

Constraint (1b) specifies that each task t in AT is assigned to a single resource only. Similar to constraint (1a), this is accomplished by summing all the x_{tri} variables for each task t and ensuring that the sum is equal to 1. Guaranteeing that the scheduled start time of

all the map tasks of each job j in J is after job j's earliest start time (s_j) is captured by constraint (2b). Constraint (2b) iterates through all the map tasks of each job j in J (stored in T_j^{mp}) and finds the x_{tri} variable for the task that is set to 1. Recall that constraint (1b) ensures that each task t has only one x_{tri} variable equal to 1. Thus, the term ($i \mid x_{tri} = 1$) identifies the scheduled start time of task t on resource r, which is at time i. Constraint (3b) enforces that the reduce tasks of all the jobs are scheduled to start only after all the job's map tasks are completed. This is accomplished by iterating through all the reduce tasks of each job j (stored in T_j^{rd}) and ensuring that the start time of the reduce task is at or after the completion time of the latest finishing map task (LFMT) of job j. The completion time of the LFMT is calculated using the *max* function as in the case of constraint (3a), discussed in the previous section.

Constraint (4b) states that N_j, which is initially set to 0, should be set to 1 if job j misses its deadline. A job j misses its deadline if the completion time of the latest finishing reduce task (LFRT) of job j is after its deadline (d_j). The completion time of the LFRT is calculated using the *max* function, as described in the previous section. To ensure that N_j is set to 1 if job j misses its deadline, the left-hand side of the constraint is set to the product of N_j and d_j, and it is asserted to be greater than or equal to the right-hand side of the constraint, which is set to the completion time of the LFRT minus d_j (see Constraint (4b) in Table 3.2). For example, given a scenario where a job j has d_j = 30 sec, and job j's completion time is 35 sec, which means the job missed its deadline. In this case, the right-hand side of the constraint evaluates to 5 sec and the left-hand side of the constraint evaluates to 0 (since N_j is initially set to 0). To satisfy constraint (4b) (i.e., make the left-hand side of the constraint greater than or equal to the right-hand side), N_j must be changed

to 1. This in turn makes the left-hand side of the constraint evaluate to 30, and the constraint will be satisfied since 30 is greater than 5.

Table 3.2. MILP Model.

$$Minimize \sum_{j \in J} N_j \qquad such\ that$$

$\left(\sum_{i \in I}\sum_{r \in R} x_{tri} = 1\right) \forall\, t \in AT$	(1b)
$\left(\left[(i \mid x_{tri} = 1)\ \ \forall r \in R,\ \ \forall i \in I\ \ \geq\ s_j\right] \quad \forall t \in T_j^{mp}\right) \quad \forall j \in J$	(2b)
$\left(\left[\begin{array}{c}(i \mid x_{tri} = 1)\ \forall r \in R,\ \ \forall i \in I\ \ \geq \\ \max\limits_{t^{mp} \in T_j^{mp}} \left((i \mid x_{t^{mp}ri} = 1)\ \forall r \in R,\ \ \forall i \in I\ \ + e_{t^{mp}}\right)\end{array}\right] \forall t \in T_j^{rd}\right) \forall j \in J$	(3b)
$\left(N_j d_j\ \geq\ \max\limits_{t \in T_j^{rd}} \left((i \mid x_{tri} = 1)\ \forall r \in R,\ \ \forall i \in I\ \ + e_t\right) - d_j\right)\ \forall j \in J$	(4b)
$\sum\limits_{t \in T^{mp}} \sum\limits_{i' \in I^*_{tri}} x_{tri'} q_t\ \leq\ c_r^{mp}$ $\forall r \in R,\ \ \forall i \in I$ where $I^*_{tri} = \{i' \mid i - e_t < i' \leq i\}$, $T^{mp} = \bigcup_{j \in J} \left(T_j^{mp}\right)$	(5b)
$\sum\limits_{t \in T^{rd}} \sum\limits_{i' \in I^*_{tri}} x_{tri'} q_t\ \leq\ c_r^{rd}$ $\forall r \in R,\ \forall i \in I$ where $I^*_{tri} = \{i' \mid i - e_t < i' \leq i\}$, $T^{rd} = \bigcup_{j \in J} \left(T_j^{rd}\right)$	(6b)
$x_{tri} \in \{0,\ 1\} \qquad \forall\, t \in AT,\ \ \forall r \in R,\ \ \forall i \in I$	(7b)
$N_j\ \in \{0,\ 1\} \qquad \forall\, j \in J$	(8b)
$i \in \mathbb{Z}$	(9b)

Ensuring that the map and reduce task capacities of each resource are not violated at any point in time is captured by constraints (5b) and (6b), respectively. Constraints (5b) and (6b) use an integer set I^*_{tri} that is defined to contain the scheduled start time of task *t,* if at time *i*, *t* is still executing on resource *r*. The purpose of I^*_{tri} is to ensure that only

tasks still executing at time *i* are included in the calculations to determine the number of tasks that are executing on a resource at time *i*. The total number of tasks executing on a resource *r*, at any point in time, must not exceed the capacity of the resource. As shown in constraints (5b) and (6b) in Table 3.2, I^*_{tri} is a set of integers defined as follows: $\{i' | \ i - e_t < i' \leq i\}$ where i' represents the values in the set I^*_{tri}. The following sample task *t1* is used to explain the use of I^*_{tri}: task *t1* has an execution time e_{t1} of 5 sec, and the decision variable $x_{tri} = 1$ has the following values for its indices: *t* is *t1*, *r* is *r1*, and *i* is 23 sec. This means that task *t1* is assigned to start executing on resource *r1* at time 23 sec. Given the values for *t1* described and the current time of interest is 25 sec, the set I^*_{tri} with the indices *t, r,* and *i* equal to *t1*, *r1*, 25, respectively (i.e., $I^*_{t1,r1,25}$) has the following values {21, 22, 23, 24, 25}. As can be observed, the set of numbers $I^*_{t1,r1,25}$ does contain the scheduled start time of *t1,* which is at time 23 sec because at time 25 sec, task *t1* is still executing on *r1*. Conversely, if the current time of interest is set to 30 sec, the set $I^*_{t1,r1,30} = \{26, 27, 28, 29, 30\}$ does not contain the scheduled start time of task *t1* (23 sec) because by that time task *t1* has already finished executing. Lastly, constraints (7b) to (9b) specify the valid domain of the decision variables, which restrict the values that the respective decision variables can have.

3.4.1 Comparison of the MILP Model and the CP Model

Overall, it is observed that the constraints in the CP Model (refer to Table 3.1) are expressed in a more intuitive and simple manner. Expressing the constraints using MILP, as shown in Table 3.2, is more complex. For example, in the formulation of the CP Model, constraint (4a) simply uses the logical operator 'implies" ($\Longrightarrow$) to set N_j to 1 if job *j* misses its deadline. Furthermore, to formulate constraint (5a) and (6a), the CP Model uses the

global constraint, *cumulative* [91]. Conversely, the formulation of the corresponding MILP Model's constraints: (4b), (5b), and (6b) requires using more complex mathematical formulas that are not as intuitive.

The fact that the MILP Model has a single decision variable for matchmaking and scheduling (x_{tri}) also makes the expression of the constraints that use the assigned (or scheduled) start time of a task more complicated (e.g., see constraints (2b), (3b), and (4b)). Conversely, the CP Model defines a decision variable for matchmaking (x_{tr}) and another decision variable for scheduling (a_t). This simplifies the formulation of the CP Model's corresponding constraints: (2a), (3a), and (4a), which can directly reference the scheduled start time of a given task t using a_t.

3.5 Design and Implementation Experience

As outlined in Section 3.2, three approaches are used to implement the CP Model and MILP Model. This section presents the experience in implementing the CP Model and MILP Model using the various software packages. Overall, it is determined that all three software packages have an associated learning curve period; however, configuring, implementing, and executing the models using LINGO and IBM CPLEX are easier compared to using MiniZinc/Gecode because both LINGO and CPLEX provide a feature-rich integrated development environment (IDE), whereas MiniZinc/Gecode only provide a command-line interface.

3.5.1 Approach 1: MILP Model Implemented Using LINGO

LINGO is a tool used to build, model, and solve optimization problems (through mathematical programs) developed by LINDO Systems Inc. [92]. LINGO provides a built-in algebraic modeling language for expressing optimization models and a powerful and

efficient solving engine capable of solving a range of mathematical optimization problems, including linear, non-linear, and integer problems.

An important feature in the implementation of the MILP Model using LINGO is captured in how constraint (4b) is implemented. LINGO provides an `If-Then-Else` flow of control construct, which performs a similar role to the `if-else` statements used in general programming languages such as Java and C. The `If-Then-Else` construct could have been used to simplify the implementation of constraint (4b) whose purpose is to set the decision variable N_j to 1 if the job j misses its deadline; however, it was determined that using the `If-Then-Else` construct to implement constraint (4b) changed the program from a mixed integer linear program (MILP) into a mixed integer *non-linear* program (MINLP). MINLPs are generally more complex and require more time to solve compared to MILPs [92], and this leads to a longer time before a solution is found. Thus, the use of the `If-Then-Else` construct is avoided in the implementation of the MILP Model. Refer to Appendix A.I for a more detailed discussion of implementing the MILP Model using LINGO.

3.5.2 Approach 2: CP Model Implemented Using MiniZinc and Gecode

In Approach 2, the CP Model is implemented with MiniZinc 1.6 [93][96], which is an open-source CP modeling language that is designed to efficiently model and express constraint programming problems. To solve the MiniZinc model, it is first converted to a FlatZinc [93] model. FlatZinc is a low-level language that is designed to be easily translated to a form which CP solving engines can use. One such solving engine that supports solving FlatZinc models is Gecode 3.7.3 (short for Generic Constraint Development Environment) [94]. Gecode is an open-source tool implemented in C++ for solving CP problems.

A novelty of the implementation of the CP Model using MiniZinc is the devising of a modified cumulative constraint for implementing constraints (5a) and (6a). The original cumulative constraint provided by MiniZinc [93] cannot be used because it is not able to handle the two different task types present in MapReduce jobs: map tasks and reduce tasks. Thus, a modified cumulative constraint, called `mr_cumulative`, is devised to ensure that map tasks and reduce tasks are only scheduled on the map task slots and reduce task slots of the resources, respectively, and to ensure that the capacities of the resources are not violated at any point in time. The required parameters for the `mr_cumulative` constraint are presented:

```
predicate mr_cumulative(array[int] of var int: startTime,
                  array[int] of int: execTime,
                  array[int] of int: resourceReq,
                  array[int] of int: resourceCapacity,
                  array[int, int] of var int: x,
                  array[int] of int: type,
                  int: taskType)
```

The first four parameters: `startTime`, `execTime`, `resourceReq`, and `resourceCapacity` are arrays that contain the start time of the tasks, the execution time of the tasks, the resource requirement of the tasks, and the capacity of the resources, respectively. These four parameters are the original parameters in the cumulative function provided by MiniZinc. The new parameters added to the `mr_cumulative` constraint include: a matchmaking variable `x` (recall Section 3.3), a `type` attribute of the tasks that indicates whether the task is a map task (`type = 0`) or a reduce task (`type = 1`), and a variable `taskType` that indicates if the constraint should be computed for map tasks (`taskType = 0`) or for reduce tasks (`taskType = 1`). Another change made in `mr_cumulative` is that it ensures the resource capacities are not violated for all the resources in R within the function, which means that `mr_cumulative` only needs to be invoked once. Conversely, the cumulative constraint

provided by MiniZinc only checks a single resource within the function, and thus needs to be invoked once for each resource.

A code snippet of the `mr_cumulative` constraint is shown:

```
forall (r in Resources) (
      forall( i in Times ) (
            resourceCapacity[r] >=
            sum( t in Tasks where type[t] == taskType) (
                  x[t,r]*resourceReq[t] *
                  bool2int( startTime[t] <= i /\ i < startTime[t] +
                              execTime[t]))
      )
);
```

The `mr_cumulative` constraint iterates through all the resources in the `Resources` set, and for each resource it ensures that at each time point in the `Times` set the capacity of the resource is equal to or exceeds the number of tasks that are running at that point in time. Note that the values of the integers in the `Times` set range from the lower bound of the task start times to the upper bound of the task completion times. The matchmaking variable, `x`, is used to ensure that only tasks mapped to the resource of interest are included in the sum. Recall that x_{tr} is 1 if task t is assigned to resource r, and each task can only be assigned to one resource. The `bool2int` library function is used to convert a Boolean value to an integer, where true is equal to 1 and false is equal to 0. The inequality that is passed to the `bool2int` function is used to ensure that only tasks that are still executing at the time of interest, i, are included in the resource capacity calculations. More specifically, a task is still running at time i if the scheduled start time of the task is less than or equal to i and i is less than the completion time of the task. A more in-depth discussion of implementing the CP Model using MiniZinc is provided in Appendix A.II.

3.5.3 Approach 3: CP Model Implemented Using CPLEX

In Approach 3, the CP Model is implemented and solved using IBM CPLEX 12.5 [15]. More specifically, CPLEX's *Optimization Programming Language* (OPL) [97] is used to implement the CP Model. OPL is an algebraic language explicitly designed for expressing optimization problems, and therefore it can provide a natural representation of optimization models that is more compact and less complex to implement compared to using general-purpose programming languages such as Java or C. The implementation of the CP Model using OPL is referred to as the *OPL Model*. The OPL Model is solved using CPLEX's *CP Optimizer* constraint programming solving engine, which provides specialized variables, constraints, and other mechanisms for modelling and solving scheduling problems efficiently [98][99]. For example, the CP Optimizer provides a built-in decision variable data type called `interval` that can be used to represent tasks (or activities) that need to be scheduled. The `interval` data type has five inherent attributes: start time, duration, end time, optionality, and intensity. The start time, duration, and end time attributes function as their names imply. The *optionality* attribute is used to indicate whether the interval is required to be present in the solution provided by the solving engine. For example, the optionality attribute can be used to model optional tasks that are not required to be executed for the solution to be valid, but can be executed if the constraints are not violated. Lastly, the *intensity* attribute defines the resource usage or utility of a task over its interval.

A key feature of the implementation of the CP Model using OPL is that it makes use of CPLEX's `interval` decision variable data type, which allows the system to use the optimized library functions and constraints that CPLEX provides, such as the `alternative`

constraint and `pulse` function [98]. This in turn allows the system to efficiently solve the CP Model by reducing processing time and memory requirements [99]. The CP Model's decision variables, a_t and x_{tr}, are implemented using CPLEX's `interval` data types as follows:

```
dvar interval taskIntervals [t in Tasks] size t.execTime
dvar interval xtr [o in Options] optional
```

The `taskIntervals` and `xtr` variables represent the CP Model's a_t and x_{tr} decision variables, respectively. Note that the keyword `dvar` is used to declare a decision variable in OPL. In the first line, the component `[t in Tasks]` specifies that `taskIntervals` is an array and there is an `interval` variable for each task in the input set `Tasks`. Each `interval` variable contains the task's start time, end time, and execution time. The `size` keyword specifies the duration of the `interval` variable, which in this case is set to the execution time of the task, `execTime`.

The second line defines that the decision variable `xtr` is an array of `intervals`, and each element in the array is associated with a tuple in the `Options` set, which is a set that contains `Option` tuples. The `Option` tuple and `Options` set are defined as follows:

```
tuple Option {
   Task task;
   Resource resource;
};
{ Option } Options = { <t,r> | t in Tasks, r in Resources };
```

The `Option` tuple represents a single x_{tr} decision variable, and it has two attributes: `Task` and `Resource`, which are also tuples themselves. The `Options` set is a derived set that contains all the possible combinations of tuples of the form `<Task, Resource>`. Going back to the declaration of the `xtr` decision variable, it is observed that the intervals contained in `xtr` are declared to be `optional`, which allows only a subset of the intervals

to be present in the solution generated by the CP Optimizer. By default, if an `interval` is not defined to be `optional`, the CP Optimizer is required to assign a start time and end time for the `interval`. Refer to Appendix A.III for a more comprehensive discussion of implementing the CP Model using IBM CPLEX.

3.6 Performance Evaluation of the Resource Management Techniques for Processing a Batch of MapReduce Jobs with SLAs

To evaluate the effectiveness and efficiency of the three approaches, simulation experiments are conducted on a closed system using various batch workloads where each batch comprises of multiple MapReduce jobs to execute. Such an experimental environment that is based on a closed system is similar to what is used by [53][71][72] and is apt for evaluating and comparing the performance of the modeling techniques and solvers. The goal of the performance evaluation is to determine which of the three approaches can solve the matchmaking and scheduling problem for a batch of MapReduce jobs with SLAs most efficiently, as well as to determine the size of workload that each approach is capable of handling.

A separate set of experiments is performed for evaluating the performance of each approach. The inputs used for a given set of experiments include a set of jobs, *J*, and a set of resources, *R*, on which to execute *J*. The MILP/CP solver program that is used by a given approach to solve the MILP/CP Model is executed on a PC that is described in Section 3.6.1. Each experiment concludes after successfully matchmaking and scheduling all the jobs in the batch. At the end of a successful experimental run the output that is produced includes the following: a schedule for the system (i.e., the assigned resource and scheduled start time for each task of each job in *J*), the time required to complete the

execution of the batch of MapReduce jobs, and the number of jobs that miss their deadlines. The processing time required by the MILP/CP solver to produce the output is measured by using the respective solver's built-in timing utilities.

The rest of this section is organized as follows. In Section 3.6.1, the experimental setup, including the metrics used in the performance evaluation, are described. Following this, a description of the system and workload parameters that are used in the experiments is provided in Section 3.6.2.

3.6.1 Experimental Setup

The simulation experiments are conducted on a PC running on Windows 7 Professional with a 3.2 GHz Intel Core 2 Duo CPU and 6.00 GB of RAM. Note that in the experiments, only the execution of the jobs on the resources is simulated. The generation and solving of the MILP Model and the CP Model are performed by executing the respective CP/MILP solver on the PC described. The performances of the three approaches are evaluated using the following metrics:

- *Batch workload completion time* (*C*): The time at which all the jobs in the batch workload finish executing.
- *Processing time overhead* (*PO*): The time required for the solver to read the input data (job, task, and resource sets), generate the model, solve the model, and produce the output.
- Number of jobs that miss their deadlines (*N*).

The values of the performance metrics, *N* and *C*, are produced as output by the simulation run. On the other hand, the value of *PO* is measured using the built-in timers of the respective software packages (LINGO, Gecode, and CPLEX). It is expected that lower

values of *PO* can be achieved and larger workload sizes can be processed if the MILP/CP solvers of the respective approaches are executed on a system with a faster CPU and more memory. Each experiment is repeated a sufficient number of times such that the confidence intervals at a confidence level of 95%, which are shown in the figures (refer to Section 3.7) as bars originating from the mean value, are less than approximately ±5%.

3.6.2 System and Workload Parameters for Batch Workloads

Table 3.3 outlines the system and workload parameters for the simulation experiments. The workloads are synthetic workloads that are generated in a similar manner as workloads used by other researchers in similar investigations. For example, the Large 2 workload is adapted from [53], whereas the other workloads are derived by using the same distributions as those used in [53]. Each workload shown in Table 3.3 is characterized by a number of parameters, which are described next. In the 'Jobs' column, n is defined as the number of jobs in the batch. The earliest start time and deadline of a job j is represented by s_j and d_j, respectively. The earliest start time of the jobs are generated using a discrete uniform (*DU*) distribution. The deadline of each job j is calculated as the sum of s_j and the product of SET_j^{max} and an execution time multiplier, *em*. Recall from Section 3.1.1 that SET_j^{max} is the maximum execution time of job j (i.e., the tasks of the job are executed sequentially on a single resource). The parameter *em* is used to determine the laxity of the job and is generated using a uniform distribution (*U*). To ensure that d_j is an integer, the *ceiling* function is used at the end of the calculation. Depending on the type of workload, the number of map tasks (k_j^{mp}) and the number of reduce tasks (k_j^{rd}) of a job j are either generated using *DU* distributions or are fixed values. The next column, 'Task Execution Times', specifies the execution times of map tasks (me) and reduce tasks (re). Since CP

does not support real values (recall the discussion in Section 3.3), the task execution times are set to be integers and various *DU* distributions are used to generate the execution times. The last column, 'Resources', defines the number of resources (m) in the resource set, R. In addition, for each resource r in R, the number of map task slots (c_r^{mp}) and reduce task slots (c_r^{rd}) is specified.

Table 3.3. System and Workload Parameters for the Batch Workloads.

Workload	***Jobs, J*** ***(s_j and d_j in sec)***	***Task Execution Times (sec)***	***Resources, R***
Small 1	$n = 5$ $s_j \sim DU(1, 50)$ $d_j \sim \lceil s_j + SET_j^{max} * U(1,5) \rceil$ $k_j^{mp} = 10, k_j^{rd} = 3$	$me \sim DU(1, 15)$ $re \sim DU(1, 50)$	$m = 10$ $c_r^{mp} = 2$ $c_r^{rd} = 2$
Small 2	$n = 5$ $s_j \sim DU(1, 50)$ $d_j \sim \lceil s_j + SET_j^{max} * U(1,2) \rceil$ $k_j^{mp} \sim DU(1,15)$ $k_j^{rd} \sim DU(1, k_j^{mp})$	$me \sim DU(1, 15)$ $re \sim DU(1, 75)$	$m = 25$ $c_r^{mp} = 2$ $c_r^{rd} = 2$
Medium	$n = 10$ $s_j \sim DU(1,50)$ $d_j \sim \lceil s_j + SET_j^{max} * U(1,2) \rceil$ $k_j^{mp} = 10$ $k_j^{rd} = 5$	$me \sim DU(1, 25)$ $re \sim DU(1, 75)$	$m = 15$ $c_r^{mp} = 2$ $c_r^{rd} = 2$
Large 1	$n = 2$ $s_1 = 0, s_2 = 500$ $d_j \sim \lceil s_j + SET_j^{max} * U(1,2) \rceil$ $k_j^{mp} = 100$ $k_j^{rd} = 30$	$me \sim DU(1, 15)$ $re \sim DU(1, 50)$	$m = 25$ $c_r^{mp} = 4$ $c_r^{rd} = 4$
Large 2 (adopted from [53])	$n = 50$ $s_j \sim DU(1, 1500)$ $d_j \sim \lceil s_j + SET_j^{max} * U(1,2) \rceil$ $k_j^{mp} \sim DU(1, 100)$ $k_j^{rd} \sim DU(1, k_j^{mp})$	$me \sim DU(1, 10)$ $re = \left\lceil \frac{\sum_{t \in T_j^{mp}} e_t}{k_j^{rd}} \right\rceil$	$m = 50$ $c_r^{mp} = 2$ $c_r^{rd} = 2$

The simulation experiments are performed using various batch workloads with different characteristics, such as the number of jobs in the batch, the number of tasks in

each job, and the execution times of the tasks, to investigate the impact of different workload characteristics on system performance. For example, in the Small 1 workload there are 5 jobs, each job with 10 map tasks with execution times varying from 1 sec to 15 sec and 3 reduce tasks with execution times varying from 1 sec to 50 sec. On the other hand, the Large 2 workload comprises 50 jobs with each job having a varying number of map tasks from 1 to 100 and a varying number of reduce tasks from 1 to k_j^{mp}. Thus, on average the Large 2 workload has about 3750 tasks and the Small 1 workload has 65 tasks.

3.7 Results of the Performance Evaluation

The following sub-sections present and discuss the simulation results of the three approaches devised to process a batch of MapReduce jobs with SLAs. Note that all three approaches focus on meeting deadlines of the jobs in the workload and their primary objective is to minimize *N*. Ensuring that *C* is small is a secondary objective that can be considered given that the primary objective is achieved. The discussion of the experimental results focus on *C* and *PO* since all three approaches are observed to be able to generate a schedule for the system that minimizes *N* (i.e., all three approaches achieve the same value of *N* for a given workload).

3.7.1 Small and Medium Workloads

Figure 3.2 and Figure 3.3 present the values of *C* and *PO*, respectively, for the three approaches when using the small and medium workloads. Note that Approach 2 is not able to generate a solution for the Medium workload even after executing the solver for a couple of hours (indicated by the missing bars in the graphs). This may be due to the limitations of the solver from being able to match make and schedule such a large number of tasks

(leading to a model that contains a large number of decision variables and constraints) on the system experimented with.

As expected, the results show that for all three approaches, the increase in the size of the workload (e.g., number of jobs and tasks) gives rise to an increase in *PO* and *C* because of the higher contention for resources. From Figure 3.3, it is observed that Approach 3 achieves the lowest *PO* (less than 0.47 sec) (note that the bars are small and may not be visible in the figure); however, it also generates a schedule that produced a slightly higher *C*. This can be attributed to the solver used in Approach 3 (IBM CPLEX) generating the first schedule that optimizes the objective function (minimizing *N*) without focusing on minimizing *C*. The lower *PO* achieved by Approach 3 is attributed to the mechanisms that CPLEX's CP Optimizer solving engine provides to efficiently solve matchmaking and scheduling problems, including the use of the interval decision variables and functions to operate on those variables [98].

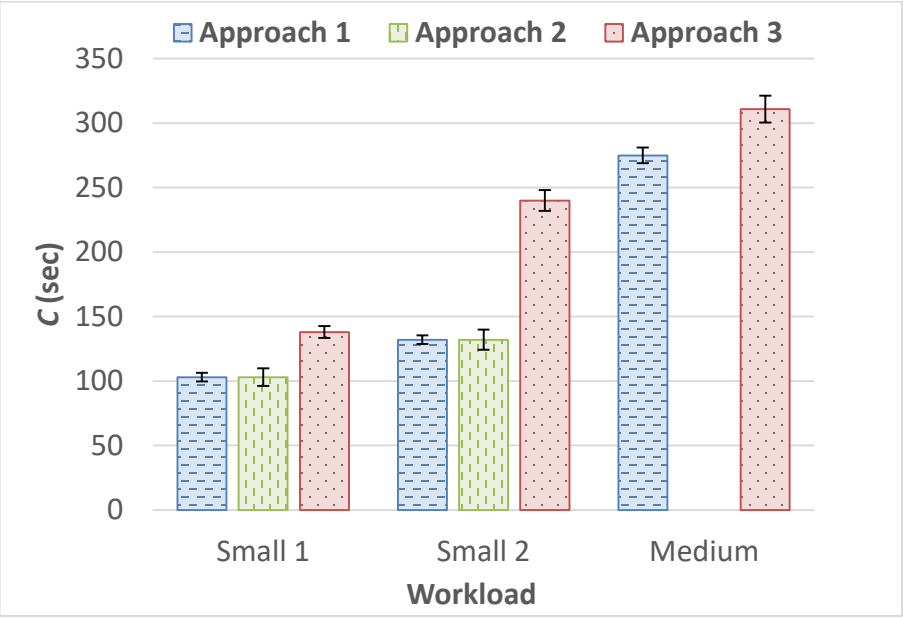

Figure 3.2. Results of *C* when using the small and medium workloads.

Another observation that is made from Figure 3.3 is that the approaches that implement the CP Model (i.e., Approaches 2 and 3) achieve a lower *PO* compared to Approach 1, which implements the MILP Model. The reason for this behaviour can be attributed to the large number of decision variables that the solver for the MILP Model has to generate and solve. Recall that the MILP Model uses a decision variable x_{tri}, and there is an x_{tri} variable for each combination of tasks in *AT,* resources in *R*, and time points in *I*. In the CP Model, there are fewer decision variables because separate decision variables are used for matchmaking, x_{tr} and scheduling, a_t.

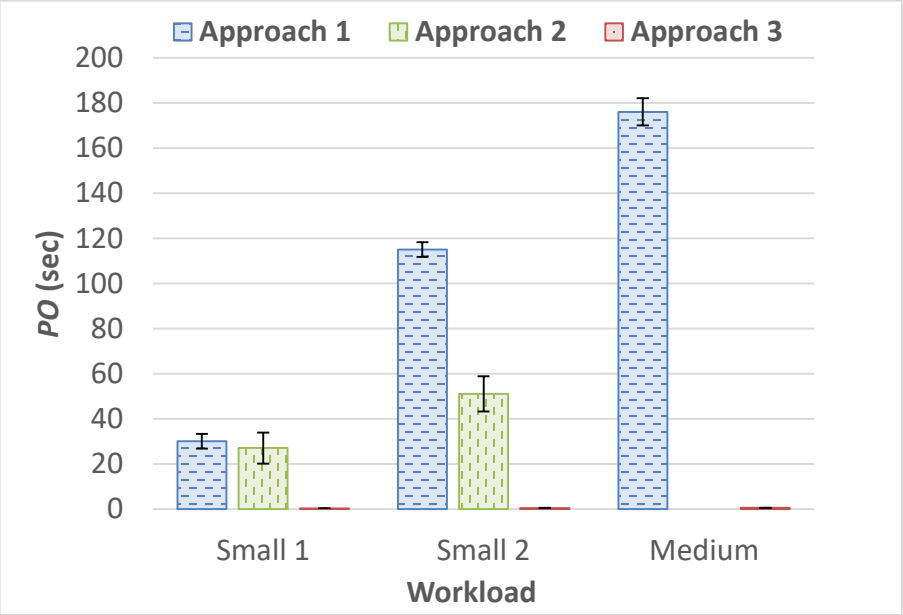

Figure 3.3. Results of *PO* when using the small and medium workloads.

3.7.2 Large Workloads

The results of *C* and *PO* when executing the large workloads are shown in Figure 3.4 and Figure 3.5, respectively. It is observed that Approach 2 is not able to handle these larger workloads (indicated by missing bars in the graphs) for the same reasons as discussed for the Medium workload. In addition, Approach 1 is not able to generate a

schedule for the Large 2 workload (indicated by missing bars in the graphs). When attempting to generate solutions for the larger workloads with Approaches 1 and 2, the system eventually ran out of memory and the solver would crash. The solvers of Approach 1 and Approach 2 cannot handle such a large number of decision variables and constraints on the system experimented with. The results demonstrate that for the Large 1 workload, Approach 3 outperforms Approach 1 in terms of *PO* and *C* for reasons similar to those discussed in Section 3.7.1.

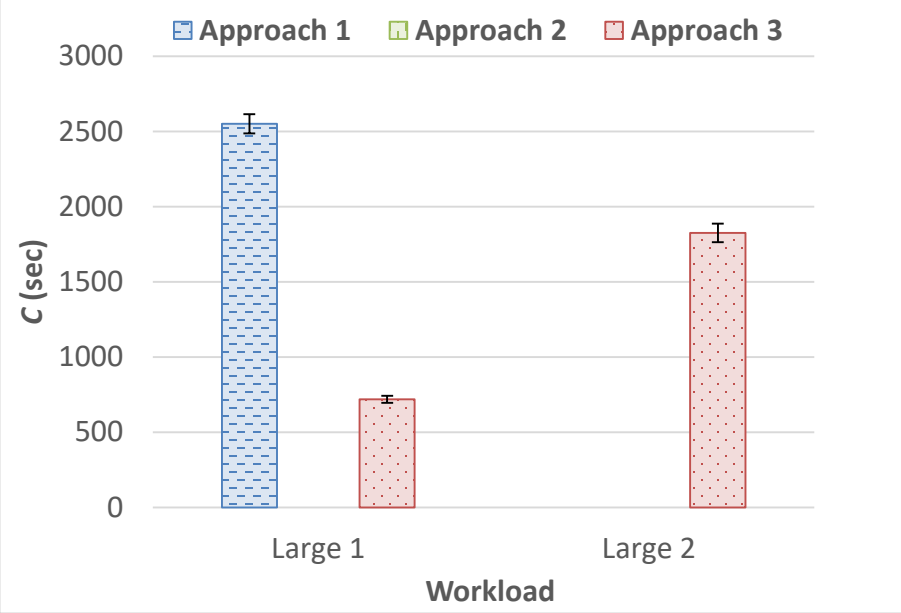

Figure 3.4. Results of *C* when using the large workloads.

To enable Approach 1 to successfully process the Large 1 workload, the granularity of *I* is reduced to decrease the number of decision variables in the model. Recall that Approach 1 uses a set of integers (*I*), which defines the range of time (or time points) when jobs can be scheduled to start executing on a resource. For example, the time range can be chosen from time 0 to the maximum completion time of the workload, which is generated by executing each job sequentially on the resources. The granularity of *I* can be changed

to restrict when jobs can start to execute. For example, the granularity of a set $I = \{1, 2, 3, \ldots, 100\}$ can be made coarser as follows: $I = \{2, 4, 6, \ldots, 100\}$. Note that such a change reduces the number of members of I by 50%. The more values in I, the longer it takes for Approach 1's solver (LINGO) to generate and solve the MILP Model because of the large number of decision variables that need to be processed. Recall that the MILP Model has a decision variable, x_{tri}, for each combination of tasks t in AT, resources r in R, and time points i in I. As such, the number of variables that are present in the MILP Model increases as the number of tasks, or number of resources, or number of time points increase.

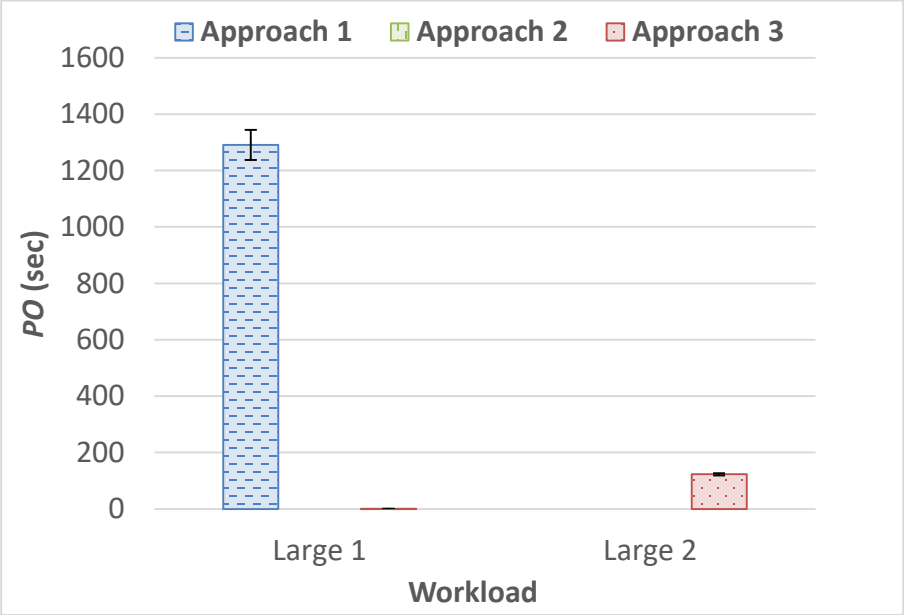

Figure 3.5. Results of *PO* when using the large workloads.

For the experiment where Approach 1 processes the Large 1 workload, the set I is set to have 100 time points with an interval of 25 sec between each time point as follows: $\{0, 25, \ldots, 2500\}$. If the granularity of I is not reduced, the MILP Model would contain a very large number of decision variables and the system would not have enough memory to find a solution and generate a schedule for the system. A disadvantage of making the

granularity of I coarser is that it can cause C to increase because some tasks cannot be scheduled to start executing at their earliest possible start times. For example, if a job j has s_j equal to 27 sec and there is an interval of 25 sec between time points, the tasks of job j cannot be executed until time 50 sec. Figure 3.4 shows that C for Approach 1 is over 2500 sec, which is about three times longer than the C achieved by Approach 3. Therefore, the results show that for Approach 1, there is a trade-off between being able to process large workloads and achieving a small C.

3.7.3 Summary of Simulation Results

This section summarizes the key observations made from analyzing the results of the experiments.

Approach 1: Approach 1 does not perform well in the experiments compared to the other two approaches. Along with Approach 2, Approach 1 did generate a schedule that produced the lowest C when using the small workloads; however, Approach 1 is measured to have a higher PO compared to Approach 2. In addition, when using the Medium workload, Approach 1 achieves a C that is 11.5% lower compared to the C achieved by Approach 3, but Approach 1's PO is also 375% higher compared to Approach 3's PO. Lastly, when using the Large 1 workload, Approach 1 is outperformed by Approach 3 in terms of both C and PO. Thus, for the system and workload parameters experimented with, it is not recommended that Approach 1 be used unless PO is not a concern. If, in addition to meeting deadlines, reducing the completion times for the batch is important, Approach 1 may be suitable to use in situations in which the matchmaking and scheduling for the jobs can be performed ahead of time (e.g., offline).

Approach 2: Approach 2 is only able to handle the smaller workloads (less than 150 tasks) on the system experimented with. When processing the large workloads, Approach 2 could not generate a schedule because the system would eventually run out of memory and the solver would crash. As discussed, along with Approach 1, Approach 2 generates a schedule with the lowest *C* when using the small workloads. Even though Approach 2's *PO* is lower compared to Approach 1's *PO*, Approach 2's *PO* is still over 100 times higher than the *PO* achieved by Approach 3. Thus, when using the small workloads, there is a trade-off between having a lower *C* (achieved using Approach 2) versus a lower *PO* (achieved using Approach 3). Similar to Approach 1, Approach 2 can be considered for processing small workloads when matchmaking and scheduling can be performed at a time prior to when the batch becomes ready to execute.

Approach 3: The experimental results demonstrate that Approach 3 has the best overall performance. Regardless of the size of the workload experimented with, it achieves a much lower *PO* compared to the other two approaches. However, when using the small workloads, Approach 3 has a slightly higher *C* compared to the other approaches. For example, when using the Small 2 workload, Approach 3 has a *C* that is 81% higher compared to the *C* achieved by Approach 1 and Approach 2. Approach 3, however, does achieve a *PO* that is over 100 times smaller compared to the *PO* measured for Approach 1 and Approach 2. On many systems satisfying the deadlines is sufficient and achieving a small batch completion time (*C*) is only a secondary objective. Furthermore, Approach 3 can process the larger workloads (i.e., Large 2 workload) that the other two approaches cannot handle. In fact, the experiments described in Section 3.7.2 indicate that Approach 3 can handle workloads containing over 1000 tasks.

3.8 Summary and Discussion

In this chapter, the problem of matchmaking and scheduling a batch of MapReduce jobs with SLAs is formulated and solved using MILP and CP. The MILP Model and CP Model that are devised are implemented and solved using three approaches:

- Approach 1: MILP Model implemented and solved using LINGO [92]
- Approach 2: CP Model implemented using MiniZinc/FlatZinc [93] and solved using Gecode [94]
- Approach 3: CP Model implemented and solved using IBM CPLEX [15].

Note that the objective of each approach is to generate a schedule that minimizes the number of jobs that miss their deadlines. Moreover, this chapter also described our experiences with using the various optimization techniques and software packages to formulate and solve the matchmaking and scheduling problem. A significant learning curve is associated with using the software in each of the respective approaches; however, configuring, implementing, and executing the models using Approaches 1 and 3 are easier compared to using Approach 2 because both LINGO and CPLEX provide a feature-rich integrated development environment, whereas MiniZinc and Gecode only provide command-line interfaces.

A number of simulation experiments are performed using various batch workloads to evaluate the performance of the three approaches. Insights into system behaviour and performance are gained from analyzing the results of the experiments, which are summarized next.

- *Superiority of Approach 3*: In all the experiments conducted, Approach 3 achieves the lowest *PO*; however, it also generated a schedule that produces a slightly

higher *C* in some experiments. In addition, Approach 3 is the only approach capable of processing the large workloads that have over 1000 tasks (refer to Section 3.7.2).

  - When using the small workloads, Approach 1 and Approach 2 achieve a lower *C* compared to Approach 3; however, the *PO* achieved by these approaches is much higher (over 100 times higher) compared to the *PO* achieved by Approach 3.

- *Superiority of CP*: The results of the experiments show that Approaches 2 and 3, which use CP, achieve a smaller *PO* compared Approach 1, which uses MILP. In addition, from the experiences in using MILP and CP to formulate the matchmaking and scheduling problem, it is found that using CP is simpler and more intuitive compared to using MILP.

Based on the results of the experiments, it is found that Approach 1 and Approach 2 are most useful in cases where the workloads are small (a few hundred tasks) and there is sufficient time to perform the resource management decisions (e.g., offline, where processing time is not a concern). On the other hand, because of its lower processing overhead, it is expected that Approach 3 can be used to devise a resource management technique that can handle an *open stream* of MapReduce jobs with SLAs. Having a low processing overhead is an important feature to consider when there is an open stream of job arrivals because a low matchmaking and scheduling overhead is key to efficiently process jobs that are continuously arriving on the system. Furthermore, Approach 3 is the only approach able to process the large workloads, comprising over 1000 tasks. Thus, a constraint programming based approach is chosen for devising a resource management

technique for processing an open stream of MapReduce jobs with SLAs, which is described in Chapter 4.

Chapter 4 MapReduce Constraint Programming based Resource Management Technique for Open Systems

This chapter concerns resource management on open systems that are subjected to a continuous stream of MapReduce jobs with SLAs arriving on the system. The experimental results from Section 3.7 showed the superiority of using the CP Model implemented using IBM CPLEX, including its more intuitive and simple formulation of constraints, lower processing overhead, and its ability to handle larger workloads. This motivated the investigation of a novel *MapReduce Constraint Programming based Resource Management* technique (referred to simply as MRCP-RM) that can effectively perform matchmaking and scheduling of an *open stream* of MapReduce jobs with SLAs. Similar to Chapter 3, the SLA for the job comprises an earliest start time, an execution time, and an end-to-end deadline. However, a key difference between the techniques described in Chapter 3 and MRCP-RM is that MRCP-RM can process an *open stream* of job arrivals whereas the techniques described in Chapter 3 can only be used in a *closed system* subjected to batch workloads with a fixed number of jobs.

The rest of the chapter is organized as follows. Section 4.1 presents an overview of the MRCP-RM technique including a discussion of the modifications made to the OPL Model. A detailed description of the MRCP-RM algorithm is then provided in Section 4.2. The performance optimizations devised to reduce the processing time overhead of the MRCP-RM technique are described in Section 4.3. Following that, the experiments conducted to evaluate the performance of the MRCP-RM technique are described in Section 4.4. A discussion of the results of the experiments are then presented in Section

4.5 and Section 4.6. Lastly, a summary and discussion of the chapter is provided in Section 4.7.

4.1 Overview of the MRCP-RM Technique

Figure 4.1 presents a diagram showing an environment deploying the MRCP-RM technique. Users submit MapReduce jobs to the system which are placed in the *job queue*. If the resource manager is available (i.e., not busy mapping another set of jobs), it invokes the MRCP-RM algorithm, which is described in more detail in Section 4.2, to perform matchmaking and scheduling (collectively called *mapping*). MRCP-RM not only maps all the newly submitted jobs in the job queue, but it also remaps the tasks of jobs that have been previously scheduled but have not started executing. This is performed to provide the most flexibility in matchmaking and scheduling to minimize the number of late jobs. For example, to minimize the number of late jobs, a newly submitted job with an earlier deadline may need to be scheduled in the place of a previously scheduled job that has a later deadline. The MRCP-RM technique uses IBM CPLEX [15] to generate an OPL Model, which is an implementation of the CP Model using IBM's Optimization Programming Language (OPL) [97]. Recall the discussion of the CP Model and the OPL Model described in Section 3.3 and Section 3.5.3, respectively. More specifically, an OPL Model is created that has new constraints added for each of the tasks that have started but not completed executing. To solve the OPL Model, MRCP-RM uses IBM CPLEX's CP Optimizer solving engine [98]. Once a solution is found, a schedule will be generated that indicates which resources that tasks should be assigned to (matchmaking) and when the tasks on a particular resource should start executing (scheduling).

An implementation of the MRCP-RM technique is developed using Java and NetBeans IDE [100]. This implementation is used to conduct the experiments described in Section 4.4. The Job, Task, and Resource entities of the resource management model (recall Section 3.1) are implemented as Java classes. The implementation of MRCP-RM also leverages the Java implementation of the IBM ILOG OPL API and IBM ILOG Concert Technology API (abbreviated Concert API) [97] to create and solve the OPL Model using the CP Optimizer solving engine.

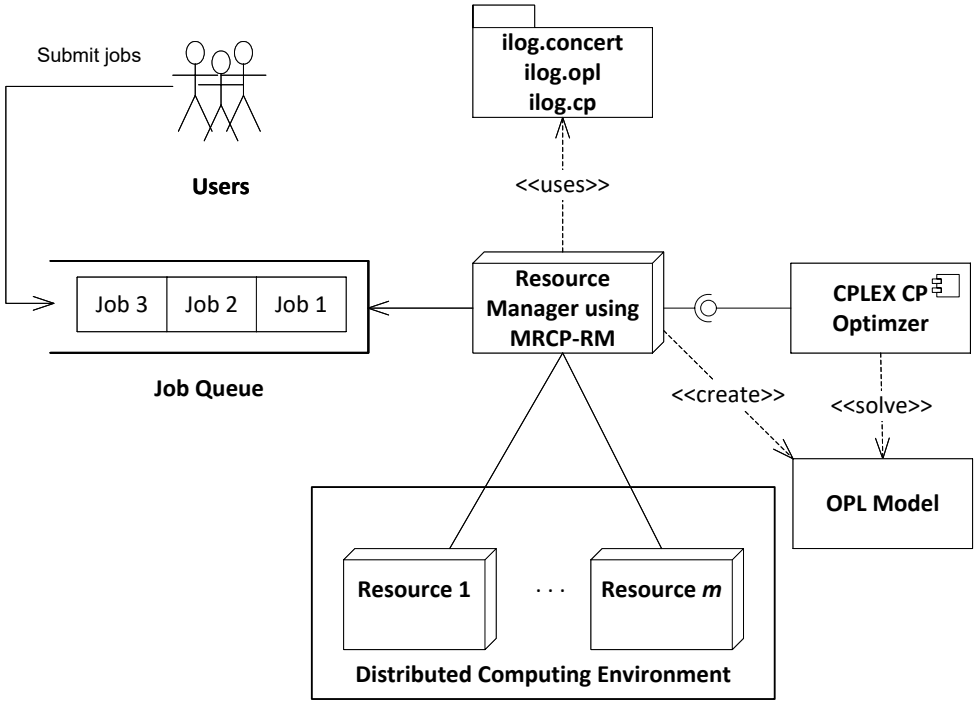

Figure 4.1. Example of a system deploying the MRCP-RM technique.

4.1.1 Modifications to the OPL Model

A few modifications to the OPL Model are made to support the MRCP-RM algorithm. The first change is the introduction of a new attribute of type integer, named `isRunning`, to the `Task` tuple. This new attribute is set to 1 to indicate that a task is currently running on a resource and cannot be rescheduled or assigned to a new resource; otherwise,

isRunning is set to 0. In addition, the implementation of constraint (2a) is modified to make use of the new isRunning attribute as follows:

```
forall (j in Jobs) {
      forall(t in Tasks: t.parentJob == j.id && t.isRunning == 0) {
          startOf(taskIntervals[t]) >= j.earliestStartTime;
    }
}
```

Recall from Section 3.3 that the purpose of constraint (2a) is to enforce that the scheduled start time of a job's tasks is after the earliest start time of the job. As shown, the constraint iterates through all the jobs in the Jobs set, and for each job, it examines each Task *t* in the Tasks set that have a value of parentJob equal to the id of the current job being examined. The *parent job* attribute of the Task tuple identifies which job the task belongs to. For example, if the parent job attribute of a task is 2, it means that this task belongs to the job with an id equal to 2. The new addition is that the constraint now only needs to check the scheduled start times of tasks that are not currently running (i.e., isRunning is 0). The tasks that are already running (i.e., isRunning is 1) already have had their scheduled start times checked previously when the tasks were initially scheduled on the system, and thus, they do not need to be checked again. Note the use of OPL's startOf() function [98] in the constraint, which returns the start time of the supplied interval variable.

4.2 MRCP-RM Algorithm

Algorithm 4.1 presents the MRCP-RM algorithm. A visual representation and a high-level overview of the algorithm in the form of a flowchart is also provided in Figure 4.2. The input required by the algorithm is a set of jobs *J* on which to map on to a set of resources *R*. The first phase of the algorithm checks the earliest start time of each job *j* in *J* to see if its earliest start time is less than the current time, and if this is true, the earliest

start time of the job is set to the current time (see lines 1-4 of Algorithm 4.1 and step 1 of Figure 4.2). The reason why some jobs have an earliest start time in the past is because these jobs were previously scheduled but have not started or completed executing.

The second phase of the algorithm checks the status/state of each task currently scheduled on the system (lines 5-6 of Algorithm 4.1 and steps 2 and 3a of Figure 4.2) to see if the task is: (1) scheduled to execute at a later time (line 7 of Algorithm 4.1 and step 4a of Figure 4.2), (2) finished executing (line 13 of Algorithm 4.1 and step 4b of Figure 4.2) , or (3) currently running (line 10 of Algorithm 4.1 and step 4c of Figure 4.2). To accomplish this, the MRCP-RM algorithm processes each resource r in R, and for each resource r, it checks when each of the tasks assigned to resource r are scheduled to execute. Note that each resource keeps its list of scheduled tasks sorted by non-decreasing order of the respective scheduled start time of the tasks. When MRCP-RM finds that a task t's scheduled start time is greater than the current time (i.e., the task has not started running), task t and the remaining scheduled tasks on the resource that t is scheduled on do not need to be processed at this point. Thus, MRCP-RM breaks out of the loop so that it can check the scheduled tasks of the next resource in R (line 8 of Algorithm 4.1).

Tasks that have started executing need to be further processed by the MRCP-RM algorithm (line 9 of Algorithm 4.1). More specifically, each task is checked to see whether or not it has completed executing. If the task has finished executing (i.e., the task's expected completion time is less than or equal to the current time) (line 13 of Algorithm 4.1), the task is marked as complete and it is removed from its parent job's task list (line 14 of Algorithm 4.1 and step 4b of Figure 4.2). Furthermore, if all the tasks of the job have finished executing, the algorithm also records that the job has completed executing (lines

15-16 of Algorithm 4.1 and steps 5-7 of Figure 4.2). On the other hand, if the task has not completed executing (i.e., the task's expected completion time is greater than the current time) (line 10 of Algorithm 4.1), the MRCP-RM algorithm adds a new constraint to the OPL Model to specify the scheduled start time, scheduled completion time, and assigned resource of the currently running task (line 11 of Algorithm 4.1 and step 4c of Figure 4.2). For example, if the first task of job 3 (denoted t3_1) is currently running on resource *r1* and has a scheduled start time and completion time equal to 11 to 30 time units, the following constraint is added to the OPL Model:

```
forall (o in Options : o.resource.id == 1 && o.task.id == "t3_1")
{
      startOf(xtr[o]) == 11 && endOf(xtr [o]) == 30;
}
```

The purpose of adding these constraints to the OPL Model is to inform the CP Optimizer of the scheduled time interval and assigned resource of the currently running tasks. This will prevent the CP Optimizer from scheduling other tasks on the resource at the same time intervals where tasks are already running (if the resource does not have the capacity to execute more than one task). In addition, the task's `isRunning` attribute is set to true (line 12 of Algorithm 4.1) to inform the CP Optimizer that the task is currently running and it does not need to enforce constraint (2a) for this task. Recall from Section 3.3 that constraint (2a) ensures that each task *t* of each job *j* in *J* has a scheduled start time that is after the earliest start time of the job. Constraint (2a) does not need to be enforced for tasks that are already executing because the scheduled start times of these tasks were already checked when they were first scheduled on the system. Moreover, since a job's earliest start time may have been changed to the current time (recall lines 1-4 of Algorithm 4.1), currently running tasks will not be able to satisfy constraint (2a) because their

scheduled start time is in the past (i.e., before the current time). Thus, as described in Section 4.1.1, constraint (2a) is changed to only check the earliest start times of tasks that are not currently running.

Algorithm 4.1: MRCP-RM Algorithm

Input: a set of jobs *J* and a set of resources *R*
Output: none

```
1:  for each job j in J do
2:      if job j's earliest start time is less than the current time then
3:          Set job j's earliest start time to the current time.
4:  end for
5:  for each resource r in R do
6:      for each task t in resource r's scheduled tasks list do
7:          if task t's start time is greater than the current time then
8:              break
9:          else //task t has started executing
10:             if task t's end time is greater than the current time then
11:                 Add a new constraint to the OPL Model that specifies task t's
                    scheduled start time, scheduled end time, and assigned resource.
12:                 Set task t's isRunning field to true.
13:             else
14:                 Record that task t is complete and remove t from its parent job's
                    tasks list.
15:                 if all the tasks in t's parent job have completed executing then
16:                     Remove the job from J.
17:             end if
18:         end if
19:     end for
20: end for
21: Create a new OPL Model and attach the data source containing J and R.
22: Generate and solve the OPL Model.
23: Extract and save the values of the decision variables (scheduled start time and
    assigned resource for each task).
```

After all the tasks are processed (line 20 of Algorithm 4.1 and step 3b of Figure 4.2), the third phase of the MRCP-RM algorithm is started. This involves using CPLEX's Java APIs to generate and solve the new OPL Model with new constraints added for each of the tasks that have started but not completed executing (lines 21-22 of Algorithm 4.1

and steps 3b and 8 of Figure 4.2). After finding a solution to the OPL Model, the values of the decision variables, which indicate the assigned resource and the scheduled start time of each task, are used to generate the new schedule for the system (line 23 of Algorithm 4.1 and step 9 of Figure 4.2). A more detailed description of how CPLEX's Java APIs [97], which includes the following packages: `ilog.concert`, `ilog.cp`, and `ilog.opl`, are used to create and solve the OPL Model is provided in Appendix B.I.

4.2.1 Complexity of the MRCP-RM Algorithm

The time complexity analysis for the MRCP-RM algorithm is described in this section. The execution time of the first phase of the algorithm (see lines 1-4) is linearly proportional to the number of jobs in *J*. Next, the execution time of the second phase of the algorithm (see lines 5-20) is proportional to the number of tasks scheduled in the system. The highest and most significant component of the execution time of the MRCP-RM algorithm comes from using CPLEX to solve the OPL Model (see lines 21-23). The overall time complexity of the MRCP-RM algorithm is thus dominated by that of the CPLEX based solution of the CP Model. In general, the complexity of solving a CP problem is NP-Complete but state-of-the art solvers can make optimizations to solve problems in polynomial time [14]. Since commercial solvers such as IBM CPLEX [15] are proprietary and their algorithms are unknown, it is difficult to theoretically define the complexity of solving the CP Model. Thus, the complexity of the algorithm is evaluated empirically through experimentation as discussed later in the performance evaluation sections. Moreover, Section 4.6.6 discusses the scalability of the algorithm.

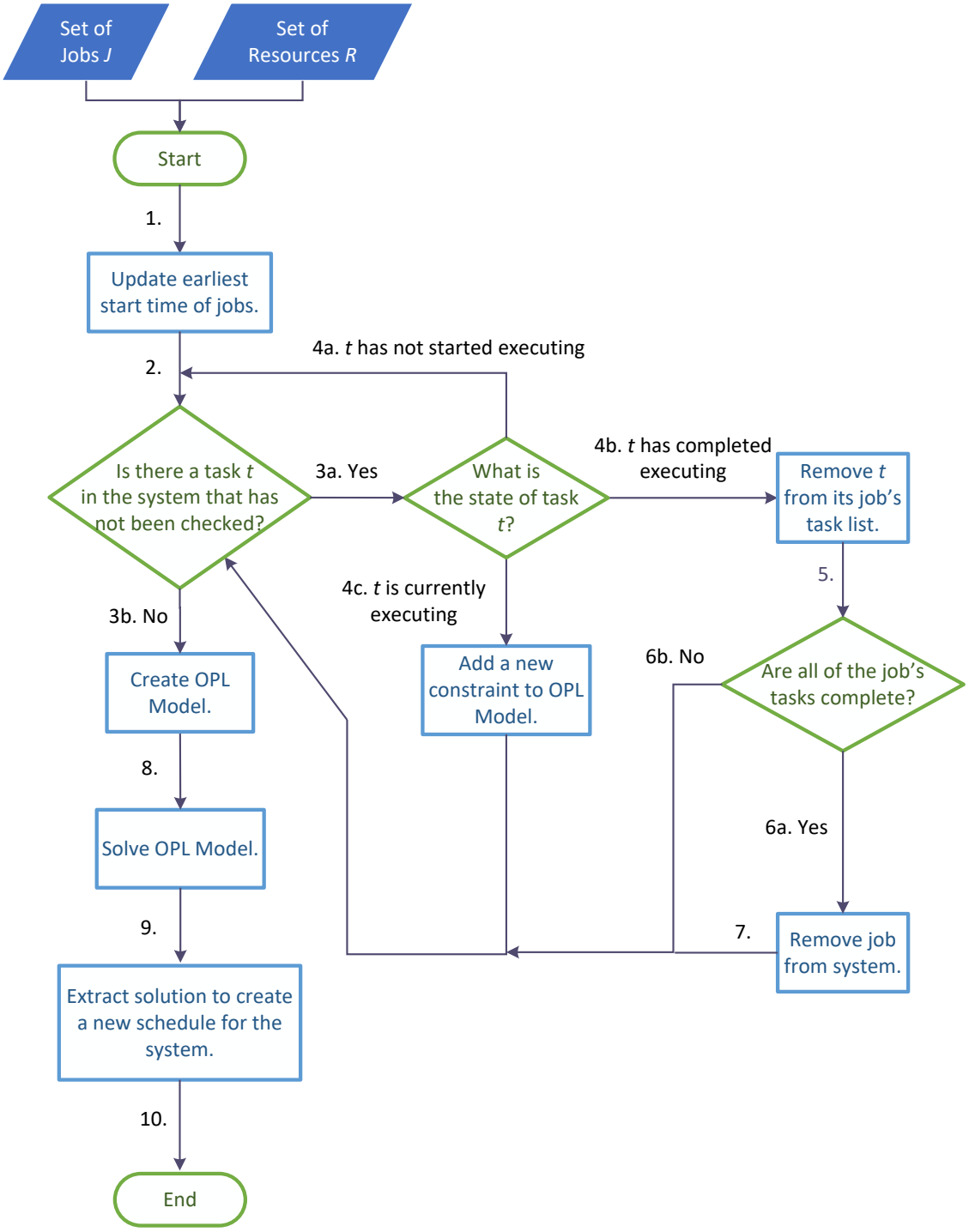

Figure 4.2. Flowchart of the MRCP-RM algorithm.

4.3 Performance Optimizations for the MRCP-RM Technique

This section describes two performance optimizations introduced to reduce the MRCP-RM technique's matchmaking and scheduling overhead, which includes the time it takes to use IBM CPLEX to generate and solve the OPL Model.

4.3.1 Performance Optimization 1: Separating the Matchmaking and Scheduling Operations

The first performance optimization (denoted *POpt1*) involves separating the matchmaking and scheduling operations. During preliminary testing, it was found that separating the matchmaking and scheduling operations in the MRCP-RM algorithm can reduce the time required for the CP Optimizer to generate and solve the OPL Model. The idea is to consolidate the number of resources in *R* into a *single combined resource*. This single combined resource contains the map task capacity (or number of map task slots) and reduce task capacity (or number of reduce task slots) of all the resources in the system. For example, consider a scenario where the system has 50 resources and each resource *r* in *R* has $c_r^{mp} = 2$ and $c_r^{rd} = 2$. In this scenario, the single combined resource *r1* has $c_{r1}^{mp} = 100$ and $c_{r1}^{rd} = 100$. For a batch of 25 jobs where each job comprises 75 map tasks and 25 reduce tasks and a system with 50 resources where each resource has 2 map task slots and 2 reduce task slots, it takes the MRCP-RM algorithm approximately 1 minute to solve the OPL Model. On the other hand, when using a single combined resource (i.e., 1 resource with 100 map task slots and 100 reduce task slots), it only takes the MRCP-RM algorithm approximately 15 seconds to find a solution to the OPL Model. This can be attributed to the CP Optimizer having fewer decision variables and constraints to process.

A description of how POpt1 is used in conjunction with the MRCP-RM algorithm is provided next. The first step is to use the MRCP-RM algorithm (described in Section 4.2) to solve an OPL Model using a single combined resource that represents all the resources in *R*. The schedule that is generated is referred to as the *single resource schedule*, and it contains the scheduled start time of each task of each job in *J*, but it does not contain matchmaking information (i.e., which resources that the tasks are assigned to). The second step is to use the *Split Single Resource Schedule* algorithm to assign the tasks from the single resource schedule to the resources in the original set of resources *R*. This is accomplished by performing the following operations. First, the algorithm moves the map tasks and reduce tasks from the single combined resource to *a set of single capacity map resources* (*MR*) and *a set of single capacity reduce resources* (*RR*), respectively. As the names suggest, each resource in *MR* has only one map task slot (i.e., $c_r^{mp} = 1$) and each resource in *RR* has only one reduce task slot (i.e., $c_r^{rd} = 1$). Second, the algorithm creates new resources to represent the original resources in *R* and assigns tasks to each of the new resources created. More specifically, each resource is assigned map tasks and reduce tasks from the single capacity resources in *MR* and *RR*, the numbers of which are equal to resource *r*'s map task capacity and reduce task capacity, respectively. A detailed discussion of the Split Single Resource Schedule algorithm is provided in Appendix B.II.

An example of invoking the Split Single Resource Schedule algorithm to partition a single combined resource that has 100 map task slots and 100 reduce task slots into 50 resources with at least one map task slot and 30 resources with at least one reduce task slot is described. First, the Split Single Resource Schedule algorithm creates 100 single capacity map resources and 100 single capacity reduce resources. The map tasks and

reduce tasks from the single combined resource are then assigned to the single capacity resources. Next, the Split Single Resource Schedule algorithm creates 50 resources, each with $\lfloor 100/50 \rfloor = 2$ map task slots, and 30 out of the 50 resources will have reduce task slots. More specifically, each of these 30 resources will have at least $\lfloor 100/30 \rfloor = 3$ reduce task slots. Since there are $100 - 30 * 3 = 10$ remaining reduce task slots, 10 out of the 30 resources will have an additional reduce task slot, and thus, these 10 resources will have a total of 4 reduce task slots. Each new resource *r* is then assigned map tasks from c_r^{mp} single capacity resources in *MR* and assigned reduce tasks from c_r^{rd} single capacity resources in *RR*.

4.3.2 Performance Optimization 2: Handling Earliest Start Time of Jobs

The second performance optimization, referred to as *POpt2*, focuses on optimizing the processing of jobs with earliest start times that are greater than their arrival times. After performing a number of preliminary experiments, it was found that when the workload comprises a large number of jobs that have earliest start times in the future (i.e., jobs that have arrived on the system but cannot start executing because their earliest start times have not yet past), the time required to perform the matchmaking and scheduling operations increases substantially. It was found that the main cause for this is that MRCP-RM maps the tasks of newly arriving jobs as well as the tasks of previously scheduled jobs that have not started executing. Recall from Section 4.1 that this is performed to provide the most flexibility in matchmaking and scheduling the jobs such that the number of late jobs is minimized. In general, when there are more tasks to map, the time required to generate and solve the OPL Model increases because there are more decision variables and constraints to process. To reduce this overhead, a mechanism is implemented to only start

matchmaking and scheduling jobs when the current time is greater than or equal to the job's respective earliest start time, s_j. Jobs that arrive and have their s_j in the future are placed in a queue of jobs that are to be mapped at a later time. This in turn prevents MRCP-RM from having to continuously map jobs that cannot execute yet, reducing the matchmaking and scheduling overhead.

4.4 Performance Evaluation of the MRCP-RM Technique

To investigate the effectiveness and efficiency of the MRCP-RM technique, an in-depth simulation-based performance evaluation using synthetic workloads is conducted. Simulation is used because it provides the flexibility to systematically change the system and workload parameters. Both synthetic and real workloads have been used by researchers in performance evaluation of resource management algorithms [101]. Although real workloads are representative of real systems, they are inflexible in the sense that they cannot be modified easily to answer "what if" questions. Synthetic workloads, on the other hand, allow researchers to directly vary the different parameters that can affect performance and thereby permit the investigation of the impact of varying a given parameter on system performance. Note that a performance evaluation of the constraint programming based resource management technique using a real workload is described in Chapter 5.

First, simulation experiments are conducted to compare the performance of MRCP-RM with that of a technique called MinEDF-WC [70] (see Section 4.4.2), which has objectives similar to the MRCP-RM technique, using a synthetic workload based on MapReduce jobs used by Facebook (referred to as the *Synthetic MapReduce Workload – Facebook*). Second, experiments are performed to investigate the effect of various system

and workload parameters on the performance of the MRCP-RM technique. A *Generic Synthetic MapReduce Workload* (described in Section 4.4.3) is used in these experiments. Note that this performance evaluation focuses on the relative performance of the MRCP-RM technique compared to that of the MinEDF-WC technique [70] and understanding the performance trends as captured in the degree of change in the performance metrics in response to changes in the system and workload parameters.

The rest of this section is organized as follows. The experimental setup and the metrics used in the performance evaluation are described in Section 4.4.1. Descriptions of the two workloads used in the simulation experiments are then described in Section 4.4.2 and Section 4.4.3.

4.4.1 Experimental Setup

The simulation experiments are executed on a PC running Windows 8 Professional 64-bit on an Intel Core i5-4670 CPU (3.40 GHz) equipped with 16 GB of RAM. The following performance metrics are used to evaluate the MRCP-RM technique:

- *Proportion of late jobs* (*P*) = $N \,/\, n$ where *N* is the number of late jobs in an experiment and *n* is the total number of jobs processed in an experiment.
- *Average job turnaround time* (*T*): The turnaround time of a job *j* is equal to $CT_j - s_j$ where CT_j is the completion time of job *j* and s_j is the earliest start time of job *j*. Thus, $T = \left[\sum_{j \in J}\left(CT_j - s_j\right)\right]/n$ where J is the set of jobs processed in an experiment.
- *Average job matchmaking and scheduling time* (*O*): A measure of the processing time incurred by executing the MRCP-RM algorithm, including the time required for generating and solving the OPL Model. $O = \left(\sum_{j \in J} o_j\right)/n$ where o_j is the

matchmaking and scheduling time of job j and is measured using Java's `System.nanoTime()`[102] method. Note that there is a distinction between the metric PO used in Chapter 3 and the metric O that is used in this chapter. PO indicates the processing overhead for an entire batch of jobs, whereas O is the average processing overhead per job executed in the open system.

In the simulation experiments, only the execution of the workload on the resources and the arrival of jobs are simulated. The MRCP-RM algorithm and the CP Optimizer solving engine are executed on the PC described earlier. Thus, O is a measured value, whereas P and T are generated as output from the simulation run. The O-by-T ratio (O/T) is used as an indicator for the processing overhead of the resource management algorithm. O/T 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).

Each simulation experiment is run long enough to ensure that the system operates at a steady state. In addition, each experiment is repeated a sufficient number of times such that the confidence intervals, at a 95% confidence level, for T and O are less than ±1% and ±5% of their respective average values for most cases. This resulted in a reasonable time for running the simulation experiments. The resulting accuracy of the simulation results is deemed to be adequate for the nature of the investigation, which focuses on examining the trend in the variation of a given performance metric in response to changes in the system and workload parameters.

4.4.2 Synthetic MapReduce Workload—Facebook

The Synthetic MapReduce Workload—Facebook is generated from workload traces collected on a production Hadoop cluster at Facebook in October 2009 [103]. This production Hadoop cluster processes and analyzes event logs from the Facebook social network for a wide variety of different applications, including business intelligence, spam detection, and ad content optimization. In addition to these production jobs that run periodically on the cluster, the cluster also processes many different experimental jobs submitted by analysts and engineers working at Facebook. These experimental jobs can include time-consuming and compute-intensive machine learning computations or smaller (1-to-2 minute) ad hoc queries that are submitted via a SQL interface. For example, such ad hoc query based jobs can include *text search* jobs and *aggregation* jobs. A text search job is used to find a specific string in the supplied input data, whereas the aggregation job is used, for example, to compute advertisement revenue from each IP address in a set of IP addresses [103].

The Synthetic MapReduce Workload—Facebook is chosen to conduct experiments in this research because it is based on a real-world MapReduce workload from a production Hadoop cluster, and it is also used by [70], which describes a resource management technique that has similar objectives to this research. As shown in Table 4.1, the Synthetic MapReduce Workload—Facebook comprises 1000 jobs and each job has a specified number of map tasks (k_j^{mp}) and a specified number of reduce tasks (k_j^{rd}). Most of the jobs (68%) in the workload have 13 or fewer tasks, and 4% of the jobs are very large and have over 1000 tasks. The execution times of the map and reduce tasks are generated using *LogNormal*, $LN(\mu, \sigma^2)$, distributions where μ is the mean and σ^2 is the variance [70]. More

specifically, the execution times of the map and reduce tasks (in milliseconds) are generated using $LN(9.9511, 1.6764)$ and $LN(12.375, 1.6262)$, respectively. Job arrivals are generated using a Poisson process and each job type has an equal probability of arriving on the system until the number of jobs that have arrived reaches its limit (see "Number of Jobs" column in Table 4.1). The earliest start time of a job j (s_j) is equal to its arrival time, and the deadline of a job j (d_j) is generated as follows: $d_j = s_j + SET_j^R * U(1, 2)$. Recall from Section 3.1.1 that SET_j^R is the time it takes to execute job j on R, assuming job j is the only job executing on R. The symbol $U(1, 2)$ represents a uniform distribution where "1" and "2" are the lower-bound (inclusive) and upper-bound (inclusive) of the distribution, respectively. In line with [70], the system used to execute this workload consists of 64 resources where each resource has one map task slot and one reduce task slot.

Table 4.1. Job Information for the Synthetic MapReduce Workload—Facebook [70].

Job Type	k_j^{mp}	k_j^{rd}	*Number of Jobs*
1	1	0	380
2	2	0	160
3	10	3	140
4	50	0	80
5	100	0	60
6	200	50	60
7	400	0	40
8	800	180	40
9	2400	360	20
10	4800	0	20

4.4.3 Generic Synthetic MapReduce Workload

Table 4.2 presents the system and workload parameters for the Generic Synthetic MapReduce Workload. This workload is adapted from [53] and is a workload that can generate jobs with a different number of tasks and different execution times. In addition, this workload provides the ability to systematically vary workload parameters such that

their effect on system performance can be investigated. The selection of parameter values and the distributions used to generate the jobs in this workload are based on [53] and [70]. In line with [70], the job arrivals are generated using a Poisson process with arrival rate, λ. The values of λ are chosen to subject the system to different levels of system load that leads to an average resource utilization on the default number of resources (refer to Table 4.2) ranging from low (approximately 5%) to high (approximately 80%). The attributes of each job j that arrives on the system are generated as follows. First, the earliest start time of job j (s_j) can be its arrival time (at_j) or a future time after at_j, depending on a random variable rv, which follows a Bernoulli distribution with parameter p. The parameter p is the probability that a job j has s_j greater than at_j. The parameter s_{max} is the upper-bound of the discrete uniform distribution (DU) used to generate the value that is added to at_j for calculating the s_j of jobs that have s_j greater than at_j. The number of map tasks (k_j^{mp}) and reduce tasks (k_j^{rd}) are also generated using DU distributions as shown in Table 4.2.

A job j's deadline (d_j) is generated as the sum of s_j and the product of SET_j^R and an *execution time multiplier, em*. Recall from Section 3.1.1 that SET_j^R is the execution time of job j when it is executed at its maximum degree of parallelism on a set of resources R with m resources. The parameter em is used to determine the laxity (or slack time) of the 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. Note that the ceiling function is used to round d_j up to the nearest integer. Moreover, the execution times of the map and reduce tasks of a job are generated using DU distributions as shown in Table 4.2. The parameter me_{max} is the upper-bound of the DU distribution used to generate the map task execution times. Note that the map task execution times include the time required to read the input data, and the

reduce task execution times include the time required to exchange data (e.g., intermediate keys) between the map phase and reduce phase. Lastly, the number of resources (m) that are used to execute the jobs in the system and the map and reduce task capacities (c_r^{mp} and c_r^{rd}) of each resource r is specified. The parameters c_r^{mp} and c_r^{rd} denote the number of map tasks and reduce tasks, respectively, that a resource r can run in parallel at a given point in time.

Table 4.2. System and Workload Parameters for the Generic Synthetic MapReduce Workload.

Parameter	***Values***	***Default Value***
Job		
Arrival rate, λ (jobs/sec)	$\lambda = \{0.001, 0.01, 0.015, 0.02\}$	$\lambda = 0.01$
Earliest start time, s_j (sec)	$s_j = \begin{cases} at_j, & rv = 0 \\ at_j + DU(1, s_{max}) & rv = 1 \end{cases}$ where $rv \sim$ Bernoulli(p) $p = \{0.1, 0.5, 0.9\}$ $s_{max} = \{1, 5, 25\} * 10^4$	$p = 0.5$ $s_{max} = 50000$
No. of Map Tasks, k_j^{mp}	$k_j^{mp} \sim DU(1, 100)$	-
No. of Reduce Tasks, k_j^{rd}	$k_j^{rd} \sim DU(1, k_j^{mp})$	-
Deadline, d_j (sec)	$d_j = \lceil s_j + SET_j^R * em \rceil$ where $em \sim U(1, em_{max})$ and $em_{max} = \{2, 5, 10\}$	$em_{max} = 5$
Task		
Map task execution time, me (sec)	$me \sim DU(1, me_{max})$ where $me_{max} = \{10, 50, 100\}$	$me_{max} = 50$
Reduce task execution time, re (sec)	$re = \left\lceil (3 * \sum_{t \in T_j^{mp}} e_t) / k_j^{rd} \right\rceil + DU(1, 10)$	-
Resource		
Number of Resources, m	$m = \{25, 50, 100\}$	$m = 50$
Capacity	$c_r^{mp} = c_r^{rd} = 2$	-

Note: DU = discrete uniform distribution, *U* = uniform distribution

The distributions used for the workload parameters: k_j^{mp}, k_j^{rd}, me, and re are adopted from [53], whereas the parameter d_j (that is not used in [53]) is generated using a similar approach to [70]. Note that the related works [53] and [70] do not consider jobs that

have s_j greater than at_j (i.e., they only considered jobs with s_j equal to at_j). This research investigates jobs with s_j equal to at_j as well as jobs with s_j greater than at_j. To examine how the individual parameters: λ, p, s_{max}, em_{max}, me_{max}, and m affect system performance, *factor-at-a-time* experiments, where one parameter is varied and the other parameters are kept at their default values (shown in the third column of Table 4.2), are conducted.

4.5 Comparison with Related Work

This section discusses the results of the simulation experiments conducted to compare the performance of the MRCP-RM technique with that of the MinEDF-WC technique [70] (recall Section 2.5.6), which has objectives similar to the MRCP-RM technique: matchmaking and scheduling an open stream of MapReduce jobs with deadlines. In order to make a valid comparison between the MRCP-RM and MinEDF-WC techniques, simulation using the Synthetic MapReduce Workload—Facebook (described in Section 4.4.2) is used to obtain the values of P and T for each technique. The same system and workload parameters used in the simulation-based performance evaluation of the MinEDF-WC technique described in [70] are also used in the simulation experiments of the MRCP-RM technique. In each simulation experiment that corresponds to a specific arrival rate, 100 simulation runs are performed that produced an interval less than ±10% of the mean value of P and an interval less than ±1.5% of the mean value of T at a confidence level of 95%. The mean value for the respective performance metric is the average computed over the 100 simulation runs.

A comparison of the MRCP-RM and MinEDF-WC techniques in terms of P and T are shown in Figure 4.3 and Figure 4.4, respectively. The results show that the MRCP-RM technique achieves a significantly lower P (up to 93% lower) and a similar T in comparison

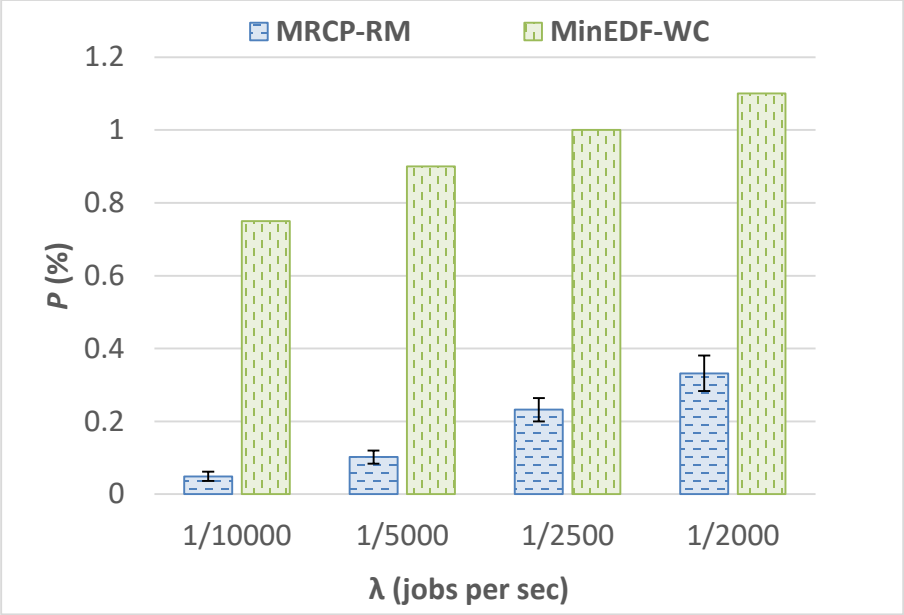

Figure 4.3. MRCP-RM vs MinEDF-WC: effect of λ on *P*.

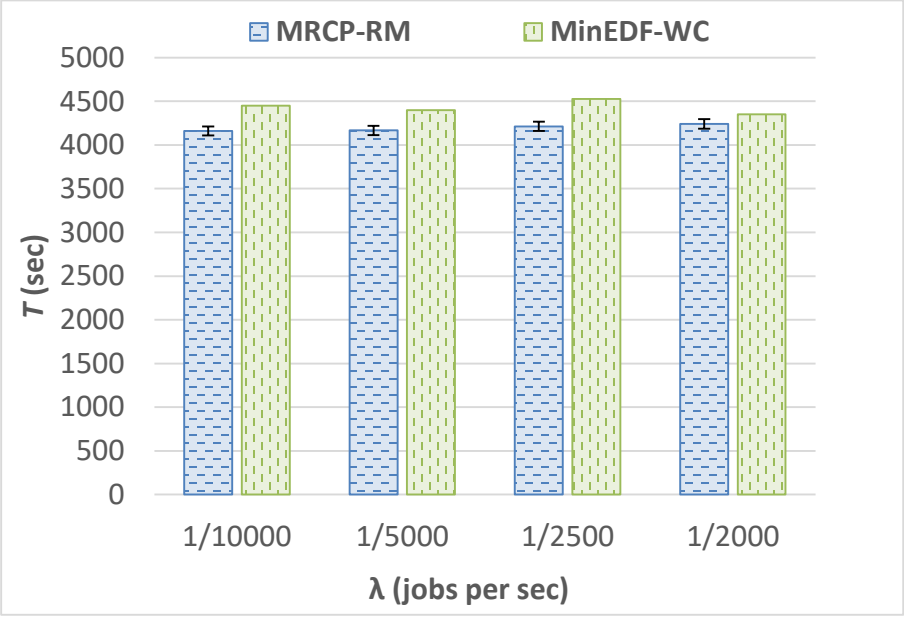

Figure 4.4. MRCP-RM vs MinEDF-WC: effect of λ on *T*.

to the *P* and *T*, respectively, achieved by the MinEDF-WC technique presented in [70]. Note that the jobs in the Synthetic MapReduce Workload—Facebook have stringent deadlines (i.e., em_{max} is 2). This means that jobs need to be executed as close as possible to

their earliest start times to meet their deadlines. The results demonstrate the superiority of the MRCP-RM technique when using the Synthetic MapReduce Workload—Facebook.

4.6 Effect of System and Workload Parameters

This section focuses on analyzing the effect of various system and workload parameters on the performance of the MRCP-RM technique (referred to simply as MRCP-RM). The experiments are conducted using the Generic Synthetic MapReduce Workload (described in Section 4.4.3) and the workload and system parameters that are investigated are outlined in Table 4.2. In the graphs presented in the following sub-sections, the confidence intervals at a 95% confidence level, which are observed to be less than ±5% of the respective mean value in most cases, are shown as bars originating from the mean value. Note that in this section, the values of *P* and *T* are shown in the same figure (see Figure 4.5, for example) with *P* displayed as a bar graph that uses the scale on the left Y-axis and *T* displayed as a sequence of points that uses the scale on the right Y-axis.

4.6.1 Effect of Job Arrival Rate

As expected, *P, T*, and *O* increase with the job arrival rate (λ) as depicted in Figure 4.5 and Figure 4.6. The increase in λ increases the rate of jobs arriving on the system, which in turn leads to a high contention for resources. The high contention for resources means that MRCP-RM is not able to schedule all the jobs to start executing at their earliest start times. This means that some jobs need to be delayed, which in turn leads to an increase in *T*, and it also can cause some jobs to miss their deadlines, increasing *P*. The reason for *O* increasing can be attributed to jobs arriving on the system more frequently causing MRCP-RM to have more jobs to match make and schedule each time it is invoked. This in turn causes MRCP-RM to have to generate and solve an OPL Model with more decision

variables and constraints, which requires more processing time and thus an increase in O is observed.

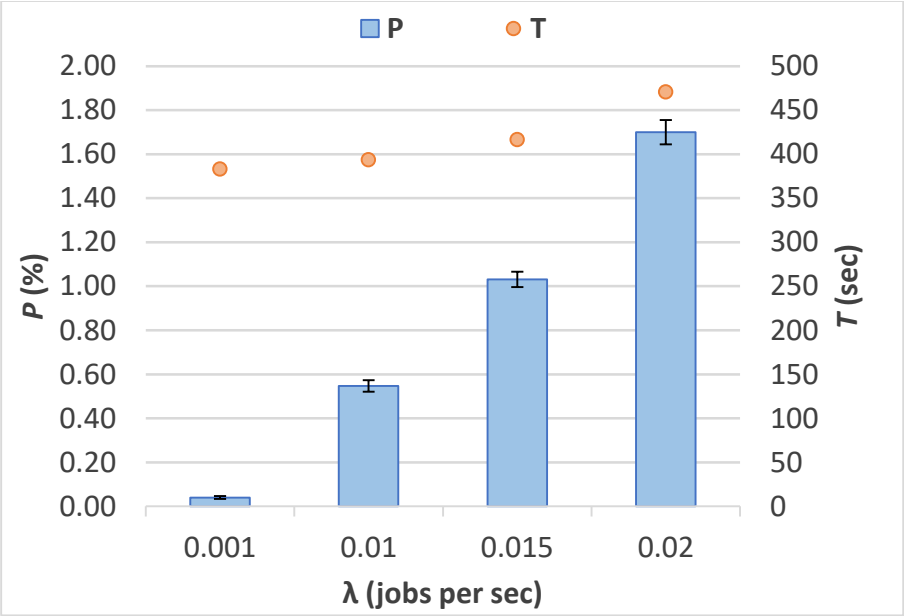

Figure 4.5. MRCP-RM: effect of λ on P and T.

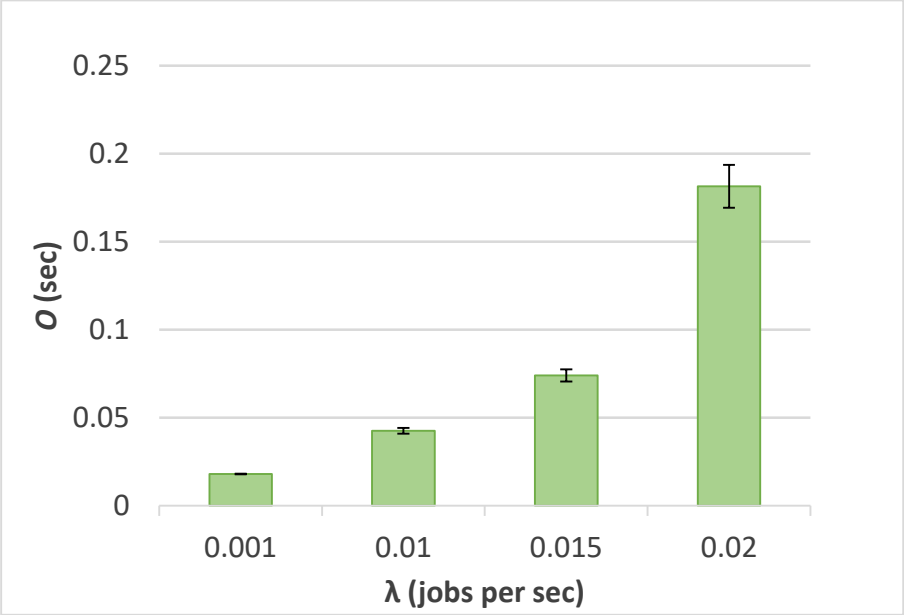

Figure 4.6. MRCP-RM: effect of λ on O.

Another reason for why O increases with λ is described. When λ is high, jobs arrive on the system more frequently, causing a situation where MRCP-RM regularly observes that there are multiple tasks that have been scheduled but not finished executing. This in turn increases the model generation and solving times because of the additional decision variables and constraints that need to be processed for each task in the system. At lower λ, this situation does not occur as often because tasks have more time to finish executing before new jobs arrive. Note that it is observed that O is still small compared to T, even at higher values of λ. For example, O/T, which is an indicator of the processing overhead, is observed to increase from 0.005% to 0.04% as λ increases from 0.001 to 0.02 jobs per sec.

4.6.2 Effect of Task Execution Times

Increasing me_{max}, the upper-bound of the execution time of map tasks (*me*), not only increases the average *me*, but it also increases the average execution time of reduce tasks (*re*). This is because of the relationship between *re* and *me* as shown in Table 4.2 . Thus, the overall execution time of a job that comprises both map tasks and reduce tasks increases as me_{max} increases. The results presented in Figure 4.7 and Figure 4.8 show that P, T, and O all increase with me_{max}. T is expected to increase with me_{max} because as me_{max} increases the required execution times of the jobs submitted to the system also increase. Furthermore, since jobs have high execution times, they remain in the system for a longer period of time, leading to a higher contention for resources and P increasing. Recall that MRCP-RM creates and solves a new OPL Model when jobs arrive and adds a new constraint to the OPL Model for each task that has started but not completed executing. In general, adding more decision variables and constraints increases the model generation and solving times, which causes O to increase. However, as captured in Figure 4.8, O is still observed to be

much smaller than T. O/T is observed to remain lower than 0.02% when me_{max} is changed from 10 sec to 100 sec.

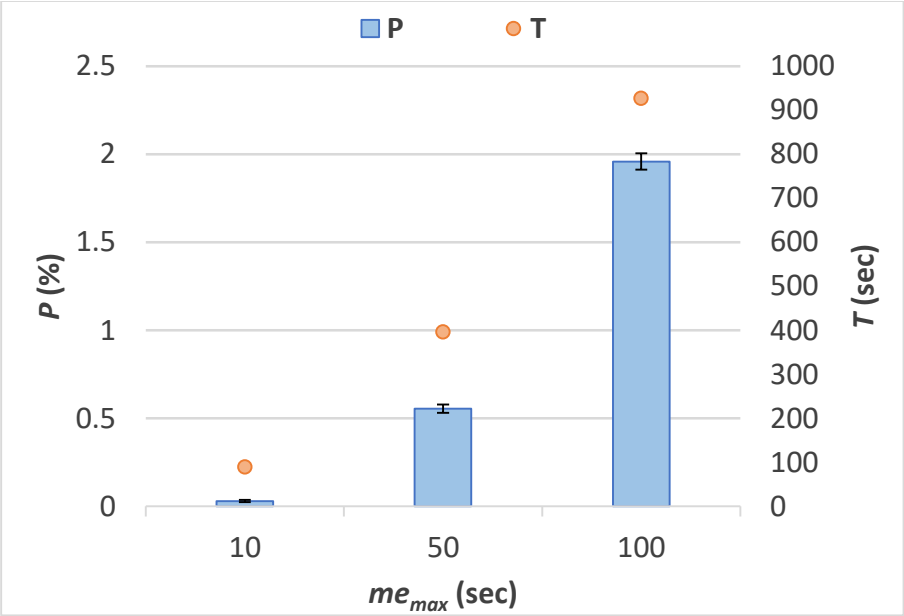

Figure 4.7. MRCP-RM: effect of me_{max} on P and T.

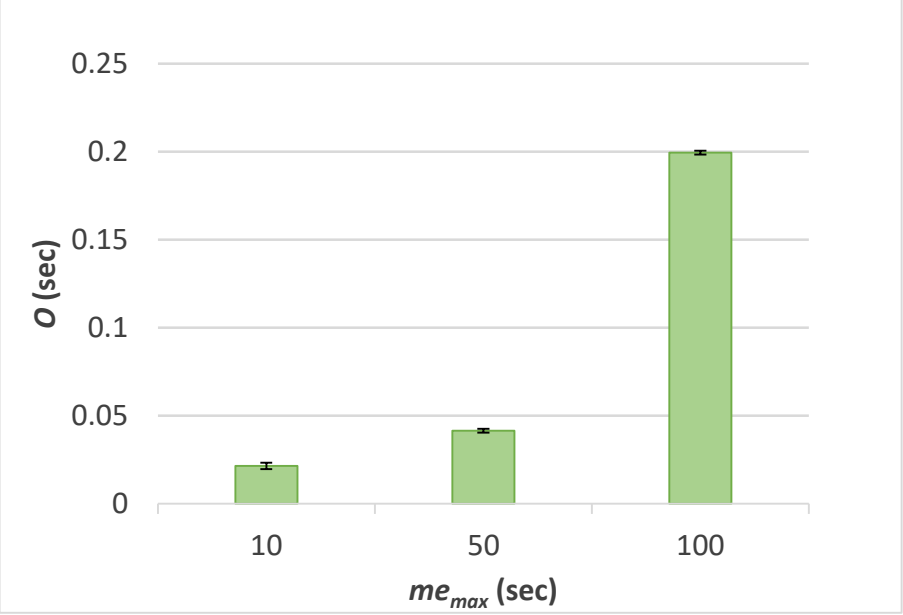

Figure 4.8. MRCP-RM: effect of me_{max} on O.

4.6.3 *Effect of Earliest Start Time of Jobs*

Figure 4.9 and Figure 4.10 show that P, T, and O tend to decrease as s_{max} increases. At higher values of s_{max}, MRCP-RM has more flexibility in scheduling jobs because the execution of jobs does not overlap as often. For instance, some jobs have earliest start times closer to their arrival times whereas other jobs have earliest start times further ahead in the future. This allows jobs to be scheduled to start executing at different points in time, which results in a lower contention for resources. In other words, more jobs can start executing at their earliest start times, resulting in a lower P and T. The low contention for resources also contributes to O decreasing because MRCP-RM has less jobs to process at a given point in time.

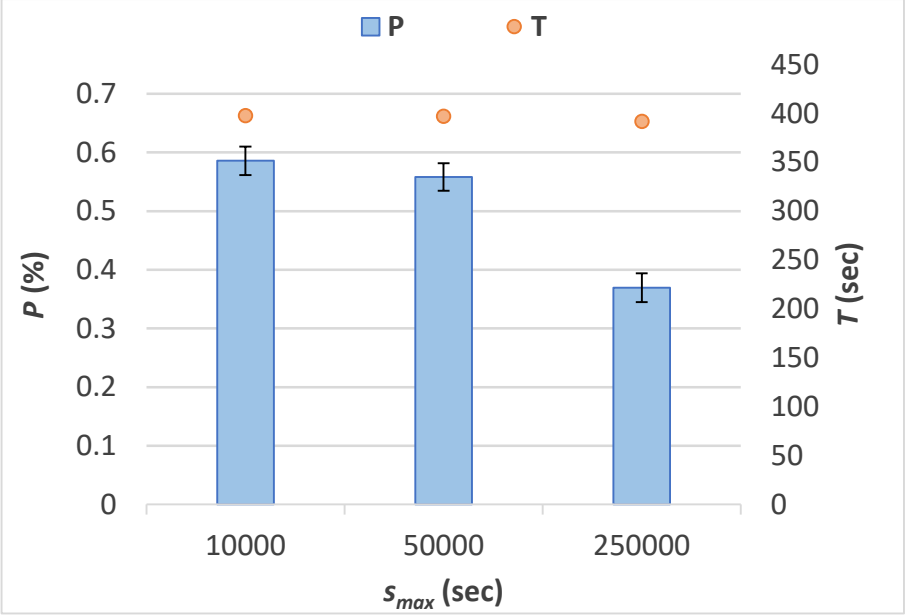

Figure 4.9. MRCP-RM: effect of s_{max} on P and T.

Figure 4.11 and Figure 4.12, in which p (the probability that a job has an earliest start time that is greater than its arrival time) is varied, shows a similar trend in performance as observed in Figure 4.9 and Figure 4.10. However, it is observed that the decrease in P

and O as p increases is not as substantial compared to the decrease in P and O as s_{max} is increased. This is because the range of earliest start times for the jobs in the experiments where p is investigated is not as large as those used in the experiments where s_{max} is investigated.

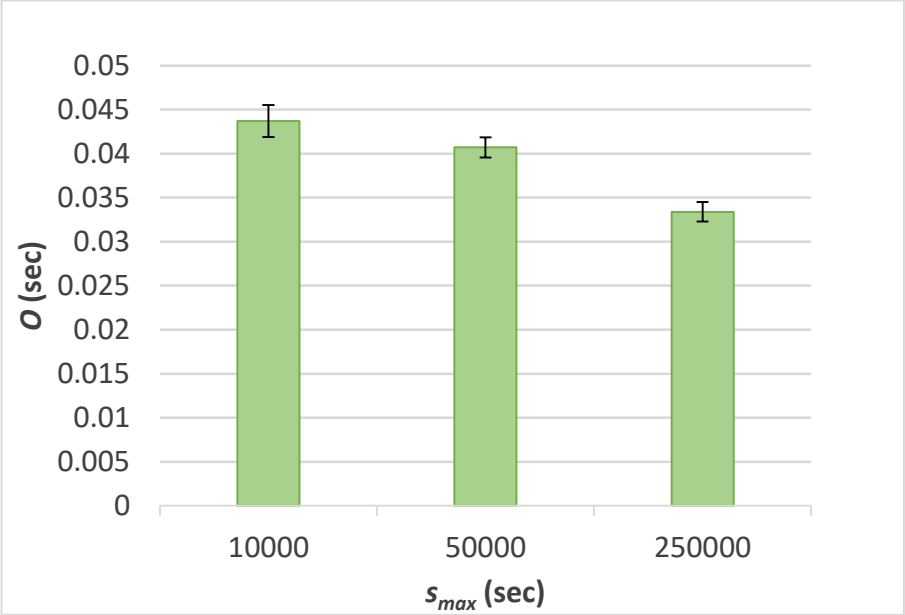

Figure 4.10. MRCP-RM: effect of s_{max} on O.

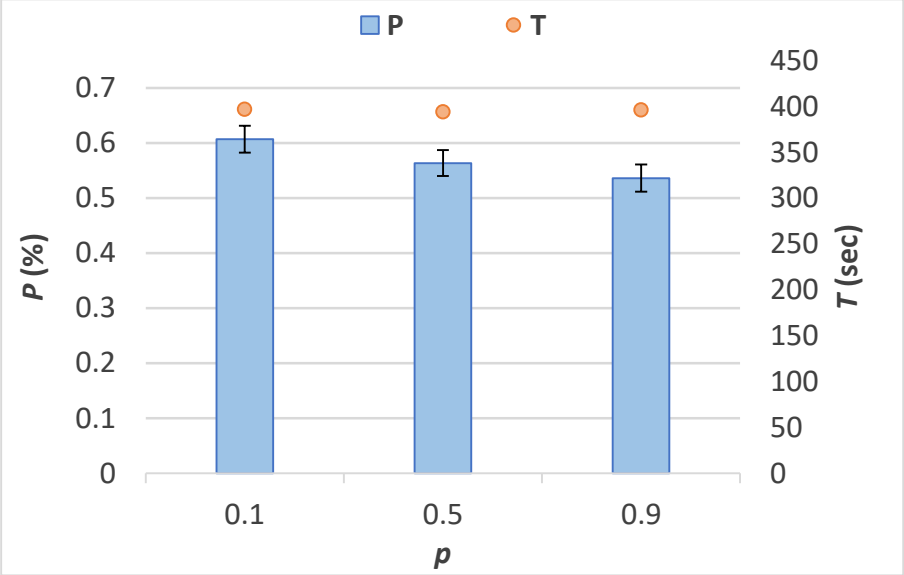

Figure 4.11. MRCP-RM: effect of p on P and T.

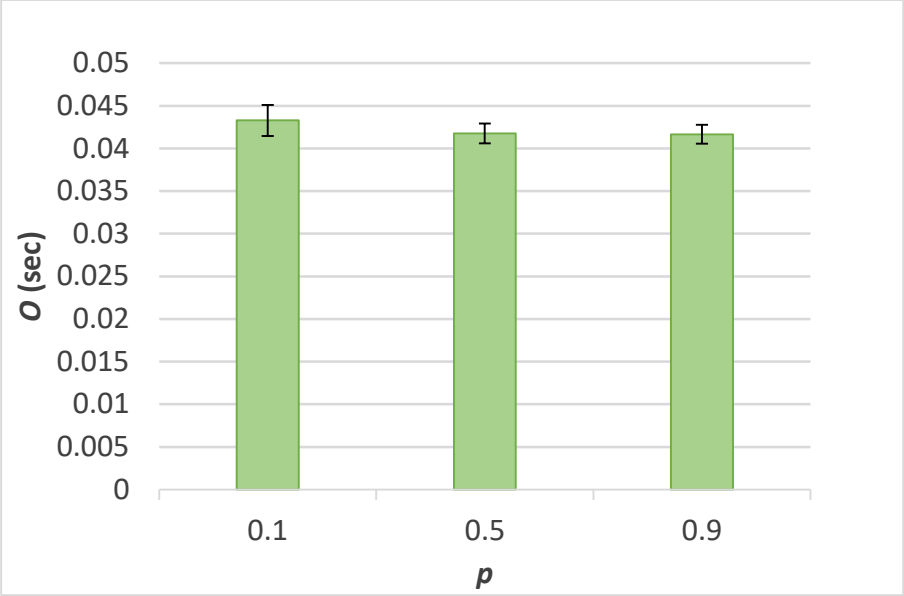

Figure 4.12. MRCP-RM: effect of p on O.

4.6.4 Effect of Job Deadlines

The impact of em_{max} on the performance of MRCP-RM is depicted in Figure 4.13 and Figure 4.14. Recall from Table 4.2 that the deadline of a job j (d_j) is calculated as $d_j = s_j + SET_j^R * U(1, em_{max})$ where s_j is the earliest start time of job j, SET_j^R is the execution time of job j when it executes at its maximum degree of parallelism on a set of resources R (comprising m resources), and $U(1, em_{max})$ is the uniform distribution used to generate the execution time multiplier for determining the laxity of the job. Thus, increasing em_{max} increases the deadlines of the jobs, leading to jobs having more laxity (or slack time). As expected, P increases with a decrease in em_{max}. This is because jobs have less slack time (tighter deadlines) and are more susceptible to miss their deadlines when em_{max} is small. Thus, in this situation jobs need to be executed at or close to their earliest start times so that they can meet their deadlines; however, since there are multiple jobs contending for

resources, some jobs cannot be executed at their earliest start times, resulting in some of these jobs missing their deadlines.

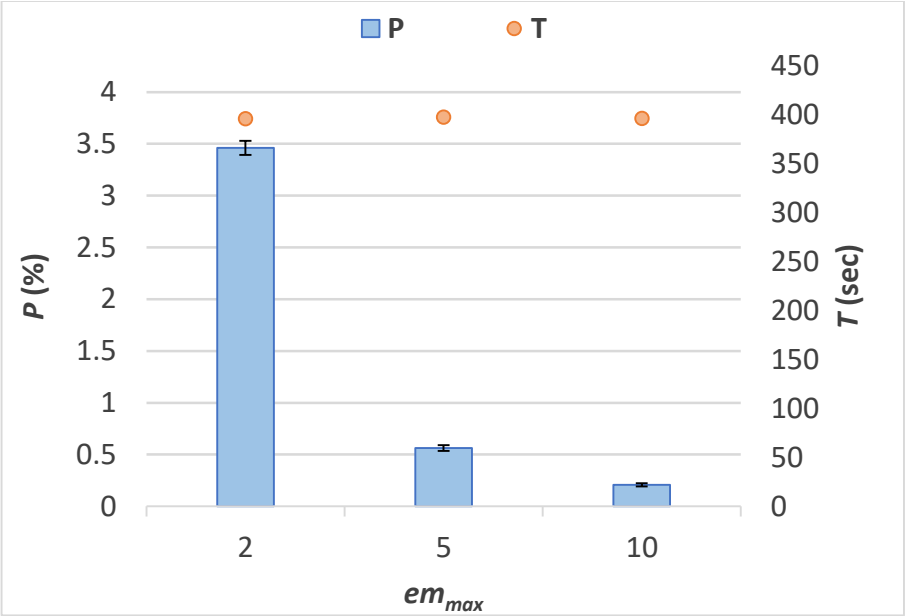

Figure 4.13. MRCP-RM: effect of em_{max} on P and T.

Moreover, as shown in Figure 4.13, it is observed that T does not significantly change when em_{max} increases. This is because for the workload and system parameters experimented with, only a small number of jobs are delayed to minimize P. A higher degree in change in T is expected if λ is increased. This will result in more jobs being present in the system and a higher contention for resources, which will lead to MRCP-RM delaying the execution of some jobs, thus increasing the turnaround time of these jobs. From Figure 4.14, it is observed that O decreases as em_{max} increases. At smaller values of em_{max} (e.g., em_{max} is 2), O is observed to be quite high because jobs have stringent deadlines, which causes MRCP-RM to require more time to perform matchmaking and scheduling of the jobs to ensure that the number of late jobs is minimized.

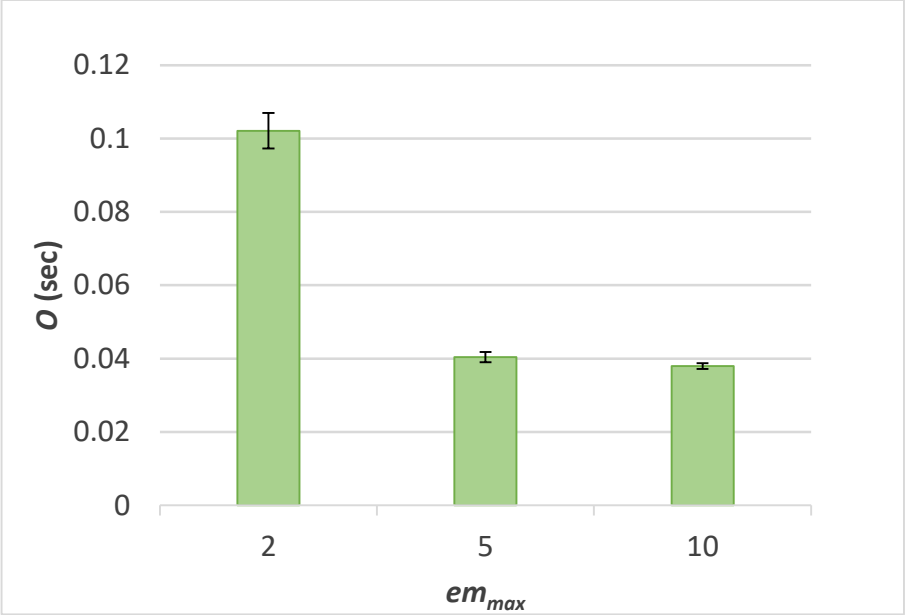

Figure 4.14. MRCP-RM: effect of em_{max} on O.

4.6.5 Effect of the Number of Resources

Figure 4.15 and Figure 4.16 show that P, T, and O increase as m, the number of resources in the system, decreases. P and T increase as m decreases because there are less resources to execute jobs (leading to a high contention for resources), which causes some jobs to be delayed for a long period of time. This prevents jobs with stringent deadlines (i.e., small laxity) from being able to finish executing before their respective deadlines. As shown in Figure 4.16, when there are fewer resources, O increases. The CP Optimizer solving engine used by MRCP-RM can find an initial feasible solution quickly, but more time is required to refine the initial solution and explore other possible feasible solutions to find a schedule that minimizes the number of late jobs, which in turn causes O to increase. More specifically, the CP Optimizer attempts to match make and schedule different combinations of tasks on the limited number of resources to see if the number of jobs that miss their deadlines can be reduced further.

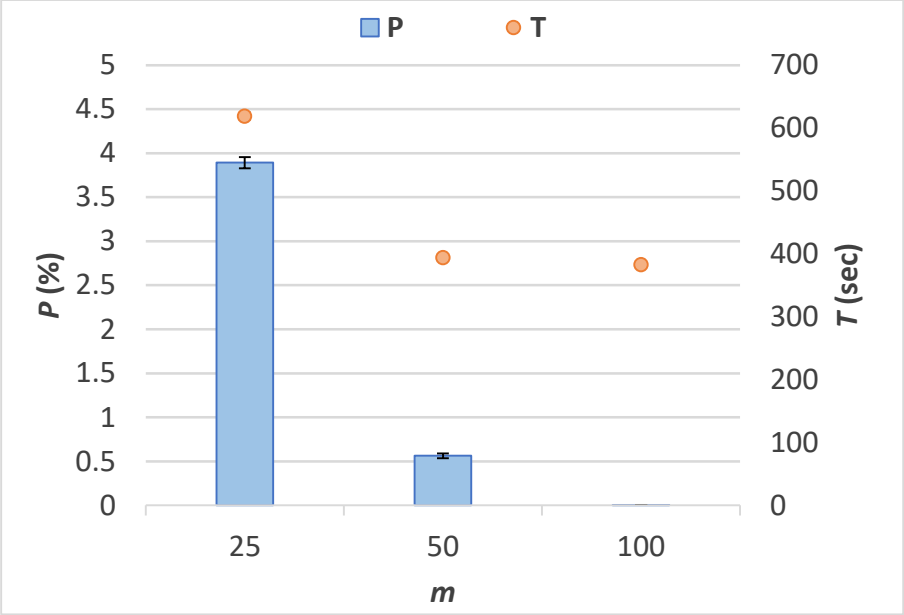

Figure 4.15. MRCP-RM: effect of *m* on *P* and *T*.

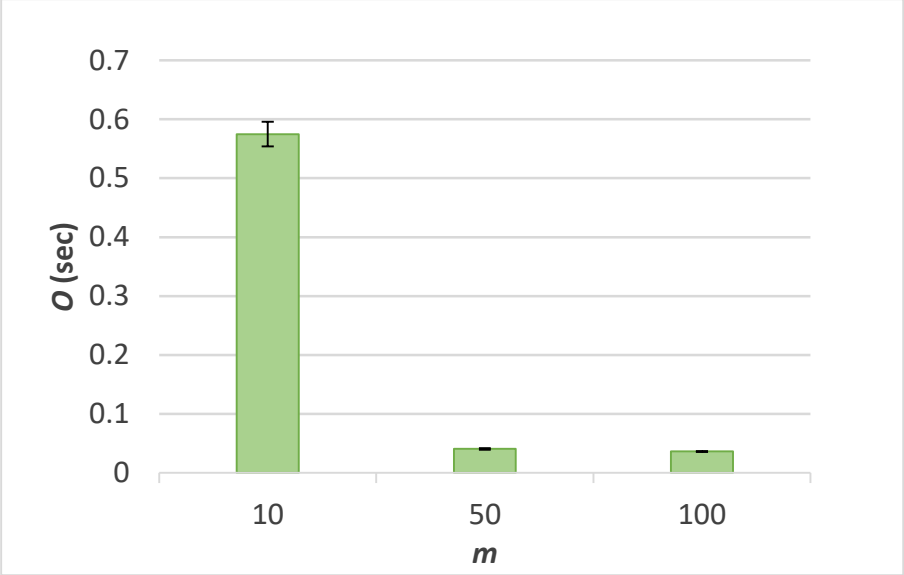

Figure 4.16. MRCP-RM: effect of *m* on *O*.

4.6.6 Scalability of the MRCP-RM Technique

Some of the experiments described in the earlier sub-sections use high arrival rates that generate a high contention for resources, leading to a resource utilization of

approximately 80% for example, to show that the MRCP-RM technique remains effective under high load (as indicated by the low values of *P* and *O/T* achieved in the results presented in Section 4.6.1 to Section 4.6.5). Recall from Section 4.4.1 that *O/T* is an indicator of the processing overhead as *O/T* 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*). In addition to performing experiments using low arrival rates (generating a low system load) and a small number of resources, high arrival rates (generating a high system load) and a large number of resources are also used to investigate the scalability of the MRCP-RM technique. The results of the experiments demonstrate that *P* remains less than 4% and the overhead of the algorithm is observed to be low: *O* remains lower than 0.57 sec and the *O/T* ratio is lower than 0.09%. Regarding the memory required for solving the OPL Model, it is observed that for λ equal to 0.001, 0.01, 0.015, and 0.02 jobs per sec, the average memory usage is 169, 489, 662, and 803 MB, respectively. This value is reasonable for today's servers, which typically have tens of GBs of RAM.

Using the experimental parameters outlined in Table 4.2 the following observations are made with respect to *O/T*. For a given arrival rate (λ), as the number of resources in the system (*m*) increases, *O/T* is observed to decrease due to less contention for resources (e.g., from 0.09% to 0.01% as *m* increases from 25 to 100) (refer to Section 4.6.5). On the other hand, for a given *m*, as λ increases, *O/T* increases because of the higher contention for resources (e.g., from 0.005% to 0.039% as λ increases from 0.001 to 0.02 jobs per sec) (refer to Section 4.6.1). These results demonstrate that *O/T* will not continuously increase if both *m* and λ increase. In other words, for a reasonable contention for resources (e.g., for a resource utilization of 0.8), *O/T* is expected to be reasonable. Moreover, if a high value

of O is a concern for a given system, it is possible to configure and tune the IBM CP Optimizer solving engine to reduce the time spent solving the OPL Model. For example, one can set a time limit for finding a solution, or allow the solver to find a sub-optimal solution which is within a given percentage of the optimal value.

4.7 Summary and Discussion

This chapter describes MRCP-RM, an effective and efficient matchmaking and scheduling technique for processing an open stream of MapReduce jobs with SLAs on a computing environment with a fixed number of resources, such as a private cluster or a set of resources acquired a priori from a public cloud. The objective of MRCP-RM is to minimize the number of jobs that miss their deadlines. To accomplish this, MRCP-RM formulates the matchmaking and scheduling problem as an optimization problem and solves it using constraint programming. An in-depth performance evaluation of MRCP-RM is conducted and a number of insights into system behaviour are gained by analyzing the experimental results as summarized next.

- *Comparison with MinEDF-WC technique* [70]: It is observed that MRCP-RM achieves a lower P and has a similar T compared to MinEDF-WC. A reduction in P as high as 93% and on average 82% is observed.
- *Effectively controlling P*: In most of the factor-at-a-time experiments conducted (refer to Section 4.6), MRCP-RM is observed to achieve a very low P (less than 0.6%). However, in the experiments where jobs are susceptible to miss their deadlines and the contention for resources is high (e.g., when em_{max} is small, or m is small, or λ is high, or me_{max} is high) P is still observed to be low: 3.46%, 3.89%, 1.7%, and 1.96%, respectively.

- *Dependence of T on resource contention*: *T* increases most significantly when there is a high contention for resources (e.g., when λ is high, or *m* is small, or me_{max} is large), resulting in some jobs not being able to start executing at their earliest start times. This demonstrates the relationship between *T* and the contention for resources.
- *Efficiency*: In the factor-at-a-time experiments described in this chapter, MRCP-RM is observed to have an *O* of less than 0.05 sec in all cases except when there is a high contention for resources. However, for these cases, *O* is still observed to be low: less than 0.57 sec.
    - The main factor that causes an increase in *O* is the time it takes for the CP Optimizer to generate and solve the OPL Model. In general, an OPL Model that has more input data (e.g., a high number of jobs, tasks, and resources) takes longer to solve due to the higher number of constraints and decision variables that need to be processed.
- *Scalability*: *O* is observed to increase when the contention for resources is high (e.g., high λ). However, *O/T* is observed to be less than 0.09% in all the factor-at-a-time experiments conducted, demonstrating that the matchmaking and scheduling overhead is small. MRCP-RM is thus observed to be scalable over the wide range of system and workload parameters experimented with. It is expected that for a reasonable range of contentions for resources, MRCP-RM can work efficiently and achieve a reasonable *O* and *O/T*.

Overall, the results of the performance evaluation demonstrate that the MRCP-RM technique can effectively and efficiently perform matchmaking and scheduling of an open

stream of MapReduce jobs with SLAs, leading to a small proportion of jobs missing their deadlines and a low matchmaking and scheduling overhead over a wide-range of system and workload parameters experimented with.

Further Improvement in Scalability: A direction for future work is the investigation of whether using *batching*: performing matchmaking and scheduling for a subset of the jobs (batch) that are currently in the queue, can be used to enhance the scalability of the MRCP-RM technique even further. It is possible that using batching can be effective when there is a very large number of jobs in the queue, leading to a large number of decision variables and constraints that require a significant amount of time for solving the OPL Model. However, using batching can also increase the processing overhead because multiple OPL Models, one for each batch, need to be generated and solved. The trade-off between the performance impacts of the increase in this processing overhead and the potential reduction in the OPL Model solving times warrants further investigation. Moreover, exploring how other resource allocation strategies, such as first-fit, worst-fit, and random fit, impact system performance when using POpt1, which currently leverages a best-fit resource allocation strategy and is described in Section 4.3.1, also forms an interesting direction for future research.

Chapter 5 Hadoop Constraint Programming based Resource Management Technique

The strong performance of the MRCP-RM technique in simulation experiments (described in Section 4.5 and Section 4.6) motivates the work presented in this chapter, which focuses on devising a revised version of the MRCP-RM technique and implementing it on a real system: Hadoop [25]. Recall from Section 2.4 that Hadoop is a popular open-source framework that implements the MapReduce programming model. In Hadoop, both matchmaking and scheduling are performed by an entity referred to as the Hadoop Task Scheduler [25]. The new technique, which is called the *Hadoop Constraint Programming based Resource Management* technique (HCP-RM), is implemented in a new scheduler for Hadoop named the *Constraint Programming based Scheduler* (abbreviated CP-Scheduler). In addition to describing the HCP-RM algorithm, this chapter describes the experiences and the challenges that are encountered in designing and implementing the CP-Scheduler in Hadoop 1.2.1. Note that Hadoop 1.2.1 is used because it was the more stable and more widely used version of Hadoop at the time this research started. However, it is possible to adapt this work to other versions of Hadoop. The CP-Scheduler is devised to perform matchmaking and scheduling of an open stream of Hadoop jobs with deadlines on a Hadoop cluster where the objective is to minimize the number of jobs that miss their deadlines. To the best of our knowledge, there is no existing research describing a CP-based scheduler for Hadoop that can perform matchmaking and scheduling of an open stream of Hadoop jobs with deadlines.

The rest of this chapter is organized as follows. Section 5.1 presents an overview of the CP-Scheduler and the HCP-RM algorithm. In Section 5.2, a description of how

matchmaking and scheduling is performed in Hadoop is provided. The focus of Section 5.3 is on the design and implementation of the CP-Scheduler. Section 5.4 describes the HCP-RM algorithm, including the technique devised to support data locality. In Section 5.5, the performance evaluation of HCP-RM, including a description of the two workloads used in the experiments, is provided. The results of the experiments are then presented and discussed in Section 5.6. The focus of Section 5.7 is on the investigation into how error in user-estimated execution times can affect system performance. With little existing work on these issues in the context of Hadoop systems, the experimental results can lead to new insights into system behaviour and performance. Lastly, in Section 5.8, a summary and discussion of the chapter is provided.

5.1 Overview of the CP-Scheduler and the HCP-RM Algorithm

Figure 5.1 shows a Hadoop cluster deploying the CP-Scheduler, which implements the HCP-RM algorithm. The Hadoop cluster comprises a single master node (NameNode and JobTracker) and *m* slave nodes (DataNodes and TaskTrackers). Recall the discussion of Hadoop provided in Section 2.4. Users submit Hadoop jobs to the JobTracker, which uses the CP-Scheduler to match make and schedule the map and reduce tasks of the jobs onto the TaskTrackers. More specifically, the HCP-RM algorithm of the CP-Scheduler uses IBM CPLEX's Java APIs to create and solve an optimization problem that models the matchmaking and scheduling problem. The optimization problem is formulated using constraint programming and it is referred to as the CP Model (refer to Section 3.3). The implementation of the CP Model using CPLEX's Optimization Programming Language (OPL) is called the OPL Model (recall Section 3.5.3).

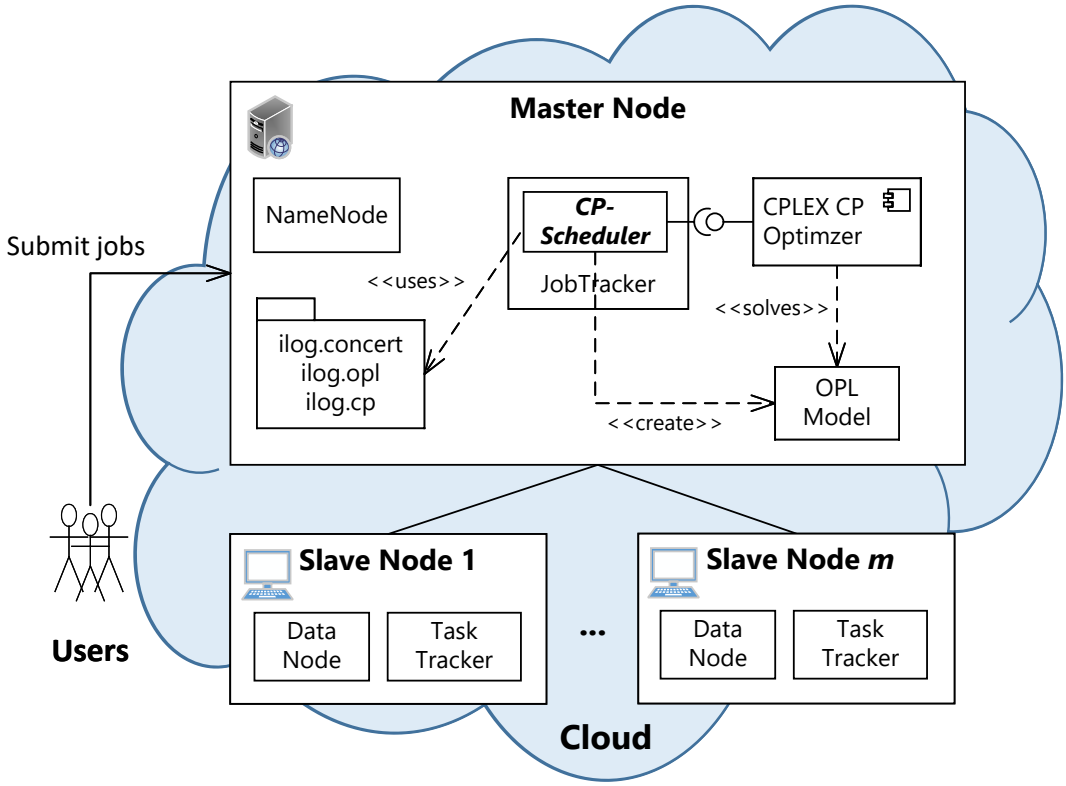

Figure 5.1. Example of a Hadoop cluster deploying the CP-Scheduler.

A flowchart that provides a high-level overview of the HCP-RM algorithm that is used by the CP-Scheduler is presented in Figure 5.2. The HCP-RM algorithm is invoked by the JobTracker each time it receives a heartbeat message from a TaskTracker (recall Section 2.4) to perform matchmaking and scheduling. The input required by the algorithm is a TaskTracker to assign tasks too. The first step is to create the input data required by the CP Model, which is a set of jobs to schedule, J, and a set of resources, R, on which J is to be executed on (step 1). Note that J includes newly arriving jobs that have not been scheduled as well as jobs that have been previously scheduled, but have not completed executing. Next, the HCP-RM algorithm checks if there are any jobs in J (step 2). If J is empty, meaning there are no new jobs to schedule and no jobs currently scheduled or executing on the system, the algorithm ends (step 3b). Otherwise, the algorithm checks to

see if there are any new jobs to schedule in *J*, or any new resources in *R* (step 3a). Note that the resources in *R* can change in two cases: (1) when new resources are added to *R* as a result of new TaskTrackers being added to the Hadoop cluster, or (2) when resources are removed from *R* because TaskTrackers that are part of the Hadoop cluster fail or crash. If there is new input data, the algorithm creates and solves a new CP Model to perform matchmaking and scheduling (step 4a). Checking for new input data in *J* and *R* is performed to prevent unnecessarily creating and solving a CP Model (which is a source of overhead) when a solution for the same input has already been found previously. In step 5, the solution of the CP Model is used to assign tasks to the TaskTrackers for execution. The algorithm then ends (step 6). A more detailed description of the HCP-RM algorithm is provided in Section 5.4.

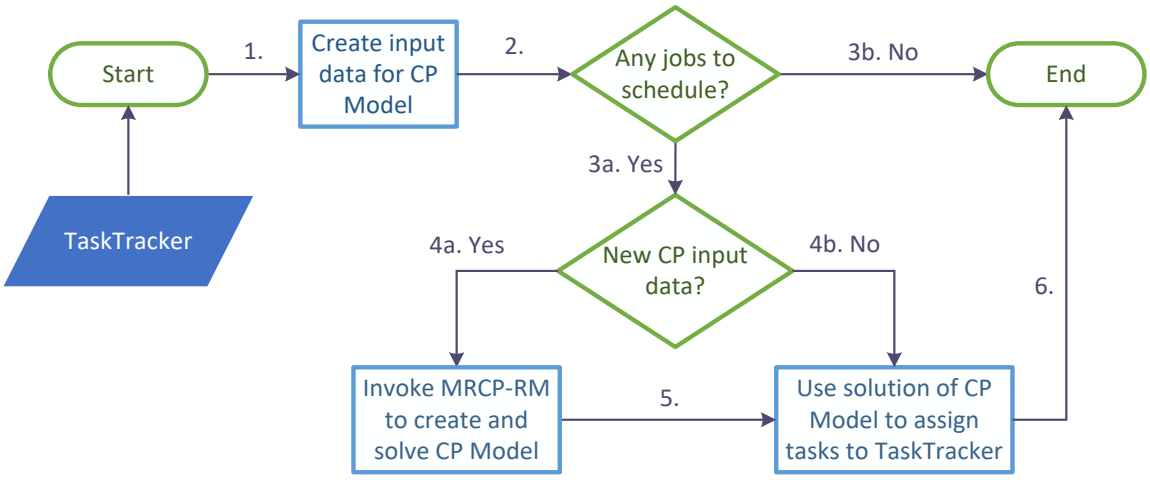

Figure 5.2. Overview of the HCP-RM algorithm.

5.1.1 Challenges in Designing and Implementing the CP-Scheduler

The challenges encountered in designing and implementing the CP-Scheduler are summarized. The main challenge is understanding the Hadoop source code to learn the

intricacies of how matchmaking and scheduling is performed in Hadoop and determining which of the Hadoop classes need to be modified to implement a new scheduler (discussed in Section 5.2). Some of the other challenges in devising the CP-Scheduler include:

- Determining how to create the input data for the OPL Model from the Hadoop classes (see Section 5.3.2)
- Integrating IBM CPLEX into Hadoop's source code (see Section 5.3.2)
- Investigating how to handle IBM CPLEX's lack of support for `long` values to represent timestamps (see Sections 5.3.3 and 5.4.2)
- Devising a technique to ensure that a specific `TaskTracker` executes the task it is assigned in the solution of the CP Model (see Section 5.4)
- Solving a problem that was discovered during preliminary testing where the reduce tasks for some MapReduce jobs stall and take a very long time to complete (discussed in Section 5.4.3)
- Determining the Hadoop classes that need to be modified to support user-specified job deadlines and to allow users to define the estimated task execution times of their jobs (described in Appendix C.I and Appendix C.II)

5.2 Matchmaking and Scheduling in Hadoop

This section provides a discussion of how matchmaking and scheduling is performed in Hadoop. Hadoop provides a pluggable scheduler framework [104] that allows developers to implement custom schedulers using their own scheduling logic and algorithms. Note that Hadoop uses the term *scheduler* to refer to the entity that performs matchmaking and scheduling. The key to implementing a custom scheduler for Hadoop is to extend Hadoop's abstract class `org.apache.hadoop.mapred.TaskScheduler` and

implement the abstract method `List<Task> assignTasks(TaskTracker tt)`. The `assignTasks()` method returns a list of tasks (including both map and reduce tasks) that the supplied `TaskTracker` should execute as soon as it receives the list. This list of tasks can be empty meaning that there are no new tasks to assign to the `TaskTracker` at the current time.

The Hadoop `org.apache.hadoop.mapred.JobTracker` class implements the Hadoop JobTracker daemon (recall Section 2.4), which is responsible for matchmaking and scheduling the MapReduce jobs that are submitted to the system. The `JobTracker` class has a `TaskScheduler` private field named `taskScheduler` which stores the reference to the `TaskScheduler` object. The `TaskScheduler` object contains the logic and algorithms used to assign and schedule tasks on to TaskTrackers. More specifically, the `JobTracker` class invokes `taskScheduler.assignTasks()` each time it receives and processes a heartbeat message from a TaskTracker. Recall from Section 2.4 that heartbeats are the periodic status messages that TaskTrackers send to JobTracker. More detail on the intricacies of matchmaking and scheduling in Hadoop is provided next with a discussion on Hadoop's default scheduler, the FIFO (first-in first-out) Scheduler.

5.2.1 Hadoop FIFO Scheduler

Hadoop's default FIFO scheduler is implemented in the `org.apache.hadoop.mapred.JobQueueTaskScheduler` class (abbreviated `JQTS`), which extends Hadoop's `TaskScheduler` abstract class. The `JQTS` class keeps jobs that are ready to execute in priority order and by default, this order is FIFO. There are two key classes used by `JQTS`: (1) `JobQueueJobInProgressListener` (abbreviated `JQ-JIPL`) and (2) `EagerTaskInitializationListener` (abbreviated `ETIL`). The `JQ-JIPL` class represents the

job queue manager, and by default, it sorts the jobs in the queue in FIFO order, but it is possible to implement a custom ordering strategy such as ordering jobs by non-decreasing order of their deadlines. `JQ-JIPL` extends Hadoop's abstract class `JobInProgressListener` (abbreviated `JIPL`), which is a class that is used by the `JobTracker` class to listen for when a job's state changes. The `JIPL` class has three key methods: `jobAdded()`, `jobRemoved()`, and `jobUpdated()`, which are invoked when `JobTracker` sees that a job is added, removed, or updated, respectively. For example, when a user submits a job to `JobTracker`, `JQ-JIPL`'s `jobAdded()` method is invoked by the `JobTracker` class to add the submitted job to `JQ-JIPL`'s queue.

The `ETIL` class prepares a submitted job for execution by initializing/creating the job's tasks, which includes creating the map tasks and assigning each one a block of data, called a *split*, to process (recall Section 2.3). A thread pool with four worker threads is deployed by the `ETIL` class to initialize jobs. Similar to the `JQ-JIPL` class, the `ETIL` class extends the `JIPL` abstract class. Thus, as soon as a job is submitted to JobTracker, `ETIL` places the submitted job into its job initialization queue called `jobInitQueue`, which by default is sorted in FIFO order. The job remains in the queue until there is a worker thread available to initialize the job.

5.3 Design and Implementation of the CP-Scheduler

This section discusses the design and implementation of the CP-Scheduler. Similar to Hadoop's default FIFO scheduler, the implementation of the CP-Scheduler starts with creating a class called `CP_Scheduler`, which extends Hadoop's `TaskScheduler` abstract class. The `CP_Scheduler` class is placed in the `org.apache.hadoop.mapred` package and a class diagram showing its key fields and methods is presented in Figure 5.3. A discussion

of the JobQueueManager and JobInitializer classes used by the CP-Scheduler is provided next. The other fields and methods shown in Figure 5.3 are described in the discussion of the HCP-RM algorithm (see Section 5.4).

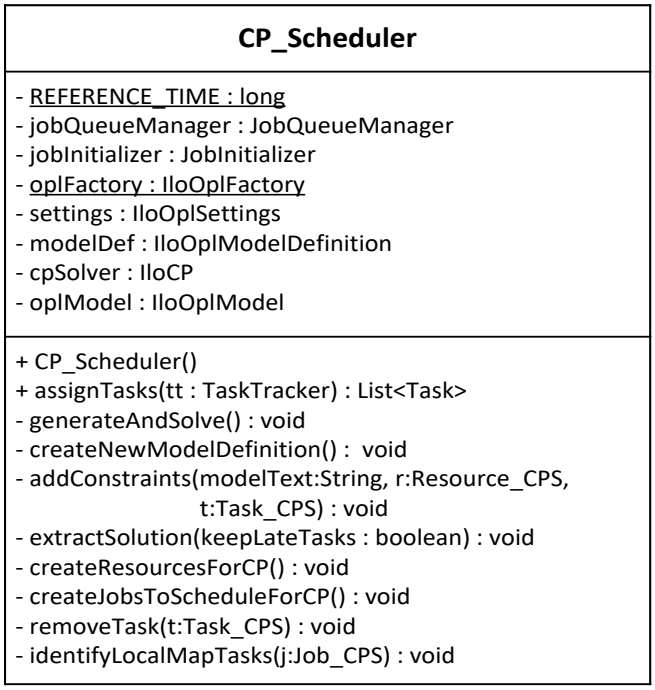

Figure 5.3. Abbreviated class diagram of the CP-Scheduler.

The JobQueueManager and JobInitializer classes extend Hadoop's JIPL class and have similar functionality to the JQ-JIPL and ETIL classes (discussed in Section 5.2), respectively. However, there are some modifications that are made to the classes, as described next. In the JobInitializer class, the resortInitQueue() method is modified to sort jobs in non-decreasing order of their deadlines (i.e., priority is given to the jobs with an earlier deadline). Moreover, the JobQueueManager's JobSchedulingInfoComparator object is also modified so that jobs with an earlier deadline will be placed at the head of the queue. Note that in Java, a Comparator is an interface used by Java collection objects to sort elements of the collection in a specified order [105]. JobSchedulingInfo is a *nested*

class [106] implemented in the `JobQueueManager` that assembles all the necessary job-related information (e.g., job id and deadline) required by the `JobQueueManager` to perform its function. In Java, a nested class is a class defined within another class. Nested classes are typically used to group together related classes and increase encapsulation.

The remainder of this section is organized as follows. First a discussion on the modifications made to the CP Model to make it work efficiently with Hadoop is presented in Section 5.3.1. A discussion of how IBM CPLEX is integrated in Hadoop is then provided in Section 5.3.2. Lastly, in Section 5.3.3, the entity classes used by the CP-Scheduler are described.

5.3.1 Modifications to the CP Model

This section discusses the modifications made to the CP Model (recall Section 3.3) to make it work more effectively in a Hadoop environment. First, a set of integer decision variables is added to the CP Model to represent the completion time of each job j in J (denoted CT_j). In addition, a new constraint is added to the CP Model to define the value of CT_j as follows:

$$\left(CT_j = \max_{t^{rd} \in T_j^{rd}} \left(a_{t^{rd}} + e_{t^{rd}}\right)\right) \ \forall j \in J$$

This constraint states that the completion time of a job j (CT_j) should be set to the completion time of job j's latest finishing reduce task.

The second modification that is made to the CP Model is to the objective function. The original objective function of the CP Model only focuses on the minimization of the number of late jobs; however, the new objective function presented here also considers

minimizing the maximum turnaround time of the jobs. The equation for the new objective function is:

$$Minimize \left(\left(\sum_{j \in J} N_j \right) + 1 \right) \times \ max_{j \in J} (CT_j - s_j)$$

The first part of the objective function aims to minimize the number of late jobs, whereas the second part of the objective function is for minimizing the maximum turnaround time of the jobs. This change is made because it was found that during preliminary experiments tasks were not being distributed evenly among the TaskTrackers (resources) of the Hadoop cluster. The net effect of the modified objective function is to minimize the number of late jobs, while also trying to distribute the tasks evenly among the TaskTrackers (i.e., perform load balancing) to lower the turnaround time of the jobs. This is confirmed to be achieved by examining the output of preliminary experiments. Note that the reason for adding 1 to the sum of N_j is to ensure that the CP Optimizer solving engine still minimizes the maximum job turnaround time if there are no late jobs in the system (i.e., sum of N_j is equal to 0).

5.3.2 Integration of IBM CPLEX with Hadoop

This section briefly discusses how IBM CPLEX [15], which is used to solve the OPL Model, is integrated with Hadoop. Note that a more detailed discussion can be found in Appendix C.III. To generate and solve the OPL Model, the CP-Scheduler imports IBM CPLEX's Java libraries, which includes the following Java APIs [97]: ILOG Concert Technology, ILOG OPL, and ILOG CP. These APIs allow the CP-Scheduler to generate and create the OPL Model and invoke CPLEX's CP Optimizer solving engine. More

specifically, the Java library packages that are used by the CP-Scheduler are named `ilog.concert`, `ilog.opl`, and `ilog.cp`.

Two additional classes are devised for the CP-Scheduler to aid in the integration of CPLEX: `OPLModelSource` and `OPLModelData`. The `OPLModelSource` class stores the source code of the OPL Model. On the other hand, the `OPLModelData` class is used by the CP-Scheduler to create the input data for the OPL Model. More specifically, `OPLModelData`, which extends the `ilog.opl.IloCustomOplDataSource` class [97], converts the `CP_Scheduler` class' `resources` and `jobsToSchedule` lists to a format that can be used by the OPL Model (i.e., generates the OPL Model's input sets: *Jobs*, *Tasks*, and *Resources*).

5.3.3 Entity Classes

Three entity classes, `Job_CPS`, `Task_CPS`, and `Resource_CPS`, are devised for the CP-Scheduler. These classes represent how the CP-Scheduler views MapReduce jobs, tasks, and TaskTrackers (resources), respectively, and store the necessary information required by the CP-Scheduler to perform matchmaking and scheduling of MapReduce jobs. An abbreviated class diagram showing the important fields, methods, and relationships of the three entity classes with the `CP_Scheduler` class is presented in Figure 5.4. As shown in the figure, the `CP_Scheduler` class maintains a list of jobs to schedule, called `jobsToSchedule`, and a list of resources to execute the jobs, called `resources`.

The `Job_CPS` class contains information required by the CP-Scheduler to map jobs onto TaskTrackers (resources). This information is retrieved from Hadoop's `JobInProgress` class and includes the following information about the job: id, earliest start time (or release time), deadline, map tasks, and reduce tasks. The `JobInProgress` class represents a MapReduce job that is being tracked by JobTracker and it stores all the

information for a MapReduce job including: the job's map and reduce tasks, its state (e.g., running, succeeded, failed), as well as accounting information (e.g., launch time and finish time). The `releaseTime` and `deadline` fields of the `Job_CPS` class store the number of milliseconds elapsed from midnight, January 1, 1970 UTC, which is known as the Unix Epoch [107]. Since the `releaseTime` field is constantly updated depending on when the job is being scheduled (discussed more in Section 5.4), `Job_CPS` has an `origReleaseTime` field that stores the release time of the job when the job is first received by JobTracker.

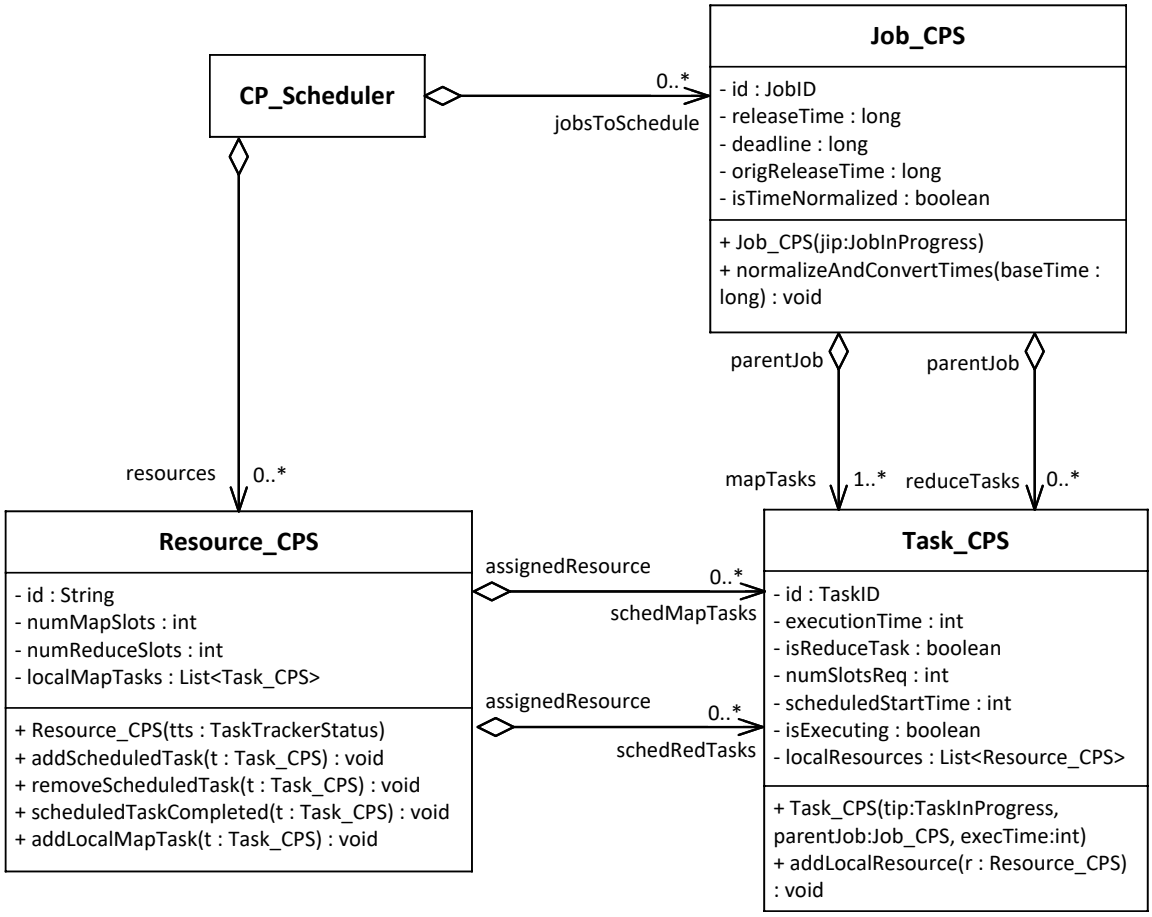

Figure 5.4. Abbreviated class diagram of the CP-Scheduler's entity classes.

The `isTimeNormalized` field indicates if the job's `releaseTime` and `deadline` fields are normalized, which involves performing the following calculations:

```
releaseTime = releaseTime – REFERENCE_TIME
deadline = deadline - REFERENCE_TIME
```

The `REFERENCE_TIME` variable is a field in the `CP_Scheduler` class that represents an epoch and stores a timestamp that is taken when the CP-Scheduler starts to map its first job. The `releaseTime` and `deadline` fields must be normalized because CPLEX does not support values of type `long`; only `int` is supported. Normalization of the `releaseTime` and `deadline` fields also includes converting the times from milliseconds to seconds in order to keep the unit of time consistent with the task execution times, which are reported in seconds. If the task execution times are reported using a different unit of time the `releaseTime` and `deadline` fields can be converted accordingly. These calculations and the time conversion is performed by invoking the `normalizeAndConvertTimes()` method.

The `Task_CPS` class stores the information that the CP-Scheduler uses for matchmaking and scheduling the MapReduce tasks including the task's: id, estimated execution time (in seconds), task type, and the number of task slots (resource capacity) required. All this information, except the estimated task execution times (discussed in Appendix C.II), is retrieved from Hadoop's `TaskInProgress` class. Once a task is mapped, its `assignedResource` and `scheduledStartTime` fields are set to the resource that the task is scheduled to execute on and the time the task is to start running, respectively. The `isExecuting` field is set to true if the task is currently executing; otherwise, it is false. The `Task_CPS` class also has a `parentJob` field that indicates the job that the task belongs to.

The `Resource_CPS` class contains TaskTracker information (retrieved from Hadoop's `TaskTrackerStatus` class), including id (hostname), the number of map task

slots, and the number of reduce task slots. The tasks that are assigned to the resource are placed in either the `schedMapTasks` list or the `schedRedTasks` list, depending on the task type. Note that both these lists keep tasks sorted by non-decreasing order of their scheduled start times. The methods `addScheduledTask()` and `removeScheduledTask()` are used to add and remove tasks from the scheduled tasks lists, respectively. The last method, `scheduledTaskCompleted()`, is invoked when a task has completed its execution. Completed tasks are moved from the scheduled tasks lists to the completed tasks lists. Note that the other fields and methods shown in Figure 5.4 that have not yet been discussed, will be described in the upcoming sections.

5.4 HCP-RM Algorithm

This section describes the *data-locality-aware* HCP-RM algorithm (see Algorithm 5.1), which is implemented in the `CP_Scheduler` class' `assignTasks()` method. The `assignTasks()` method is invoked by JobTracker each time it receives a heartbeat message from a TaskTracker and its purpose is to assign tasks to the TaskTracker for execution. Since MapReduce/Hadoop applications typically process a large amount of data, frequent transmission of data from one machine in the Hadoop cluster to another machine in the cluster over the network can deteriorate system performance due to network delays and limited bandwidth in the cluster. Thus, it is beneficial to use a data-locality-aware system that can limit the data transfer between nodes as much as possible. A data-locality-aware system assigns tasks to execute on nodes that contain (or are close to) the input data of the task to eliminate (or minimize) data transmission over the network. The technique that is devised to support data locality is described in Section 5.4.1 and is used by the HCP-RM algorithm.

Algorithm 5.1: HCP-RM Algorithm

Input: TaskTracker *tt*

Output: list of tasks for the supplied TaskTracker to execute, named *assignedTasks*

```
1:  Get currently available map task slots and reduce task slots of tt.
2:  call createResourcesForCP() returning hasNewResources
3:  call createJobsToScheduleForCP() returning hasNewJobs
4:  if CP-Scheduler's jobsToSchedule list is empty then return empty list
5:  if hasNewJobs = true or hasNewResources = true then
6:      call generateAndSolve()
7:  end if
8:  res ← Get Resource_ CPS object from CP-Scheduler's resources list with the same
    id as tt.
9:  for each available map task slot in tt do
10:     while there is a map task scheduled on res do
11:         Task_CPS t ← Get the map task with the earliest scheduled start
                         time from res.
12:         tip ← t.getTaskInProgress()
13:         if tip is complete then
14:             call removeTask()
15:         else
16:             jip ← t.getParentJob().getJobInProgress()
17:             call jip.obtainSpecificMapTask(tip) returning mapTask
18:             Add mapTask to assignedTasks.
19:             break
20:         end if
21:     end while
22: end for
23: for each available reduce task slot in tt do
24:     while there is a reduce task scheduled on res do
25:         Task_CPS t ← Get the reduce task with earliest scheduled start time
                         from res.
26:         tip ← t.getTaskInProgress()
27:         if tip is complete then
28:             call removeTask()
29:         else if t.getParentJob().getMapTasks().isEmpty() then
30:             jip ← t.getParentJob().getJobInProgress()
31:             call jip.obtainSpecificReduceTask(tip) returning reduceTask
32:             Add reduceTask to assignedTasks.
33:             break
34:         end if
35:     end while
36: end for
37: return assignedTasks
```

A walkthrough of the HCP-RM algorithm is provided next. The input required by the algorithm is a TaskTracker to assign tasks to, and the output is a list of tasks for the supplied TaskTracker to execute (includes both map and reduce tasks). The first step (line 1) is to calculate the number of currently available map task slots and reduce task slots of the supplied TaskTracker by subtracting the number of running tasks from the capacity of the resource as shown:

$$\texttt{availMapSlots} = \texttt{mapCapacity} - \texttt{runningMaps}$$

$$\texttt{availReduceSlots} = \texttt{reduceCapacity} - \texttt{runningReduces}$$

The next step (lines 2-3) is to create the `Resource_CPS` list (called `resources`) and `Job_CPS` list (called `jobsToSchedule`), which are required as input to the OPL Model. The `createResourcesForCP()` method is invoked to create the `resources` list. More specifically, the `createResourcesForCP()` method uses the `JobTracker`'s `activeTaskTrackers()` method to retrieve a collection of `TaskTrackerStatus` objects, which are then used to create `Resource_CPS` objects via its constructor (refer to Figure 5.4). If the `createResourcesForCP()` method finds that the `resources` list has at least one new resource added or at least one resource has been removed, it returns true; otherwise, false is returned. Moreover, the `jobsToSchedule` list is created by invoking the `createJobsToScheduleForCP()` method, which checks the `JobQueueManager`'s `jobQueue` (a collection of `JobInProgress` objects) for new jobs that are ready to run (i.e., setup is complete and tasks are initialized) and creates a new `Job_CPS` object for each new job using the `Job_CPS`' constructor. In addition, `createJobsToScheduleForCP()` invokes the `identifyLocalMapTasks()` method, which is described in detail in Section 5.4.1, to identify the resources that store the input data of each map task. This is required for making

the HCP-RM algorithm data-locality-aware. Note that the `jobsToSchedule` list includes new jobs to schedule and jobs that have been previously scheduled but have not completed executing. If there are new jobs added to the `jobsToSchedule` list, `createJobsToScheduleForCP()` returns true; otherwise, false is returned.

The next step is to check if the CP-Scheduler's `jobsToSchedule` list is empty. If this condition is true, then an empty task list is returned (line 4). Otherwise, the algorithm continues and if either the `hasNewJobs` or the `hasNewResources` flags are true, the CP-Scheduler's `generateAndSolve()` method (described in Section 5.4.2) is invoked to create and solve an OPL Model (see lines 5-7). These two flags are used to prevent unnecessarily invoking `generateAndSolve()`, which is a source of overhead, when a solution to the OPL Model for the same input data (`jobsToSchedule` and `resources`) has already been found. Once there is a solution to the OPL Model, either a new solution or a previously generated solution, the next step is to retrieve the assigned map and reduce tasks for the supplied TaskTracker. This is accomplished by retrieving the supplied TaskTracker's corresponding `Resource_CPS` object from the CP-Scheduler's `resources` list (line 8), which is the `Resource_CPS` object that has the same id as the supplied `TaskTracker`. The retrieved `Resource_CPS` object is saved in a variable named `res`.

Each available map task slot of the supplied `TaskTracker` is then assigned a map task to execute if there is one available (lines 9-22). This is accomplished by performing the following operations. First, the map task (a `Task_CPS` object) with the earliest scheduled start time is retrieved from `res` and saved in a variable *t* (line 11). The map task *t*'s corresponding `TaskInProgress` (abbreviated `TIP`) object is then retrieved as shown in line 12. The status of task *t* is then checked to see if it has completed executing using the

retrieved `TIP` object. If this is true, the CP-Scheduler's `removeTask()` method is invoked (lines 13-14) to remove the task from the system. In addition, `removeTask()` also checks if the job's `mapTasks` and `reduceTasks` lists are empty (i.e., checks if the job has completed executing). If this is true, the job's `releaseTime` is set to its `origReleaseTime`, and the job is moved from the CP-Scheduler's `jobsToSchedule` list to the `completedJobs` list. On the other hand, if the map task has not completed executing, the task is assigned to the supplied TaskTracker `tt` for execution (lines 15-20). This is accomplished by invoking a new method named `obtainSpecificMapTask(TaskInProgress tip)` that is implemented in Hadoop's `JobInProgress` class. As the name suggests, given a `TaskInProgress` object, the `obtainSpecificMapTask()` method returns the corresponding Hadoop `Task` object (i.e., `Task` object that has the same id as `TaskInProgress`). The `Task` object that is returned is then added to the `assignedTasks` list.

The same logic captured in lines 9-22 is then applied to the TaskTracker's reduce task slots (see lines 23-36), except with one change to the `else` statement (line 15). The `else` statement is changed to an `else if` statement that checks if all the map tasks of the job have completed executing before the job's reduce tasks are scheduled (line 29). This needs to be performed because of a problem discovered during preliminary testing that is described in Section 5.4.3. Moreover, a new `obtainSpecificReduceTask()` method is implemented in Hadoop's `JobInProgress` class that returns the reduce task (`Task` object) with the same id as the supplied `TIP` and is used as shown in line 31. Lastly, the `assignedTasks` list, which now contains the tasks that the supplied `TaskTracker` should execute, is returned (line 37).

5.4.1 Technique to Support Data Locality

This section describes the heuristic technique devised for the HCP-RM algorithm to support data locality. The technique increases the estimated execution time of *non-local tasks* (which are tasks that execute on a resource that does not store the input data for the tasks) by an estimated time taken to transfer the data from a resource containing the input data of the task. This allows the system to know that a task has a lower estimated execution time on a resource where its input data is stored locally compared to a resource that does not have its input data stored locally. This should therefore make it more likely that jobs are assigned to resources which contain their input data.

The HCP-RM algorithm uses a method called `identifyLocalMapTasks()` (see Algorithm 5.2) to identify, for each map task in the supplied job, the resources that each map task can execute on locally. A resource can provide *local execution* if the input data of the task is stored locally on its disk. As described earlier, `identifyLocalMapTasks()` is invoked by the `createJobsToScheduleForCP()` method (line 3 of Algorithm 5.1). The `createJobsToScheduleForCP()` method invokes `identifyLocalMapTasks()` multiple times: once for each job in the `jobsToSchedule` list. A walkthrough of `identifyLocalMapTasks()` is provided next.

The input required by the method is a `Job_CPS` object, which is an object that represents how the HCP-RM algorithm views a MapReduce job that has been submitted to the system. The method processes each of the supplied job's map tasks (line 1), and for each map task *t* (represented by a `Task_CPS` object), the following operations are performed. First, the resources (represented by `Resource_CPS` objects) where *t*'s input data is located are retrieved and then saved in a list called `localResources` (line 2). This

operation is implemented by invoking the `getSplitLocations()` method of Hadoop's `TaskInProgress` class, which returns the hostnames of the TaskTrackers (resources) where a task's input data is stored. Next, the method iterates through each resource *r* in `localResources` (line 3) and performs two operations. The first operation is to add the map task *t* to resource *r*'s `localMapTasks` list (line 4), which is a list that stores all the map tasks that *r* can execute locally. The second operation is to add *r* to *t*'s `localResources` list (line 5), which is a list that stores all the resources that can execute *t* locally. After all the map tasks of the input job are processed the method ends.

Algorithm 5.2: CP-Scheduler's *identifyLocalMapTasks()*

Input: Job_CPS *job*
Output: none

```
1:  for each map task t in job do
2:      localResources ← Get the resources where t's input data is located.
3:      for each resource r in localResources do
4:          r.addLocalMapTask(t)
5:          t.addLocalResource(r)
6:      end for
7:  end for
```

The OPL Model (recall Section 3.5.3) that is solved by the HCP-RM algorithm is modified to allow the execution time of a map task to be specified on a per resource basis. Recall that the OPL Model uses a data type called `tuple` for grouping together related data. More specifically, the OPL Model defines `Job`, `Task`, and `Resource` tuples (described in detail in Appendix A.III) to represent MapReduce jobs, MapReduce tasks, and resources (TaskTrackers), respectively. The OPL Model also defines a tuple called `Option` to represent the x_{tr} decision variable, which is a binary variable used for matchmaking (recall Section 3.3). The variable x_{tr} is set to 1 if task *t* is assigned to execute on resource *r*; otherwise, x_{tr} is set to 0. Similar to the x_{tr} variable, the `Option` tuple contains two attributes:

a `Task t` and `Resource r`. A set of `Option` tuples named `Options` is derived to contain all the possible combinations of tuples in the form `<Task, Resource>`. The change that is made to the OPL Model to allow the execution time of a map task to be specified on a per resource basis is removing the *execution time* attribute from the `Task` tuple and moving the attribute to the `Option` tuple. The execution time attribute for the `Option` tuples are read from a two-dimensional array called `TaskExecutionTimes[Task, Resource]`, which specifies the execution time for each task t in AT on each resource r in R.

The `TaskExecutionTimes` array is populated by the `OPLModelData` class. Recall from Section 5.3.2 that the `OPLModelData` class is used by the HCP-RM algorithm to convert the `resources` and `jobsToSchedule` lists to a format that the OPL Model can read (i.e., generates the OPL Model's input datasets: `Jobs`, `Tasks`, and `Resources`). If a map task t can execute on a resource r where the input data is stored locally on its disk (referred to as *local execution*), the estimated execution time is equal to the execution time of the map task specified by the user (e_t). Conversely, if the map task executes on a resource where the input data is not stored locally (referred to as *non-local execution*), the estimated execution time is the sum of e_t and the time required to transfer the input data to the resource (called the *data transmission time*). Note that the HCP-RM algorithm knows whether a task t can execute on a resource r locally by checking the data in t's `localResources` list or r's `localMapTasks` list, which are initialized by the `identifyLocalMapTasks()` method described earlier.

5.4.2 Generate and Solve Method

The CP-Scheduler's `generateAndSolve()` method is invoked by the HCP-RM algorithm to generate and solve the OPL Model (see line 6 of Algorithm 5.1). A

walkthrough of `generateAndSolve()`, presented in Algorithm 5.3, is provided next. The first step of the method is to check the value of the CP-Scheduler's `REFERENCE_TIME` variable (abbreviated `RT`). If `RT` is not already initialized (i.e., `RT` is equal to -1), `RT` is set to the current system time, and in addition, the `oplCurrentTime` variable is set to 0 (lines 1-3). Recall that `RT` is required to normalize the `Job_CPS`' `releaseTime` and `deadline` fields as discussed in Section 5.3.3. If `RT` is already initialized (i.e., `RT` is not equal to -1), then `oplCurrentTime` is set to the current time minus `RT` and the `oplCurrentTime` is converted into seconds (lines 4-7). The `oplCurrentTime` variable represents the current time in the view of the OPL Model. IBM CPLEX does not support values of type `long`, and thus, the current time value must be referenced from an epoch, which in this case is `REFERENCE_TIME`.

In the next steps (lines 8-12), each job (represented by a `Job_CPS` object) in the CP-Scheduler's `jobsToSchedule` list has its `releaseTime` and `deadline` fields normalized with respect to the `REFERENCE_TIME` and converted to seconds by invoking the `Job_CPS`' `normalizeAndConvertTimes()` method (recall Section 5.3.3) (line 9). In addition, jobs that have a `releaseTime` less than the `oplCurrentTime` have their `releaseTime` updated to `oplCurrentTime` because a job cannot start executing in the past (lines 10-11). In line 13, a new OPL Model definition object is created by invoking the CP-Scheduler's `createNewModelDefinition()` method, which is described in detail in Appendix C.IV. This method adds a new constraint to the OPL Model for each task in the system that has started executing but has not finished. This is needed for specifying that the task's assigned resource is occupied from the interval starting from the task's scheduled start time to its scheduled completion time. After a new model definition object is created, a new OPL

Model with input data from the CP-Scheduler's `jobsToSchedule` and `resources` lists is then generated and solved using IBM CPLEX (lines 14-15).

Algorithm 5.3: CP-Scheduler's *generateAndSolve()*

Input: none
Output: none

```
1:  if REFERENCE_TIME = -1 then
2:      REFERENCE_TIME ← System.currentTimeMillis()
3:      oplCurrentTime ← 0
4:  else
5:      oplCurrentTime ← System.currentTimeMillis() – REFERENCE_TIME
6:      Convert oplCurrentTime to seconds.
7:  end if
8:  for each job j in CP-Scheduler's jobsToSchedule list do
9:      call j.normalizeAndConvertTimes (REFERENCE_TIME)
10:     if oplCurrentTime > j.getReleaseTime() then
11:         j.setTempReleaseTime(oplCurrentTime)
12: end for
13: call createNewModelDefinition()
14: Create a new OPL model and attach the data source containing the
    CP-Scheduler's jobsToSchedule and resources list.
15: Generate and solve the OPL model.
16: call extractSolution()
```

After a solution to the OPL Model is found, the assigned resource and scheduled start time of each task is retrieved and saved to its corresponding `Task_CPS` objects' `assignedResource` and `scheduledStartTime` fields, respectively, by invoking the CP-Scheduler's `extractSolution()` method (line 16). In addition, the tasks (`Task_CPS` objects) that are assigned to resource *r* (a `Resource_CPS` object) are added to *r*'s scheduled map tasks list or scheduled reduce tasks list depending on its task type. The discussion provided in Appendix B.I describes in more detail how the steps shown in lines 14-16 are performed. In addition, Appendix B.I also discusses the purpose of the `CP_Scheduler` class' `factory`, `settings`, `cpSolver`, `modelDef`, and `oplModel` fields.

5.4.3 Stalling Problem for Reduce Tasks

During preliminary testing, it was found that in some situations the reduce tasks of a job would be scheduled to execute on the system, but the reduce tasks would stall (i.e., would execute partially and then stop executing) and take a very long time to complete. It was discovered that the reason why the reduce tasks were not being executed in a timely manner is because by default Hadoop starts to schedule/execute reduce tasks of a job once a few of its map tasks have finished executing (i.e., Hadoop does not wait until all the job's map tasks have completed before scheduling/executing the reduce tasks). This can be problematic because if not all the map tasks of the job are completed, the reduce tasks also cannot finish executing, and thus the reduce tasks remain idle and unnecessarily consume the reduce task slots of TaskTrackers. This can in turn delay the execution of the reduce tasks of jobs that already have their map tasks completed if there is no other available reduce task slots to execute on.

Note that it is possible for the reduce tasks of a job *j* to stall for a long period of time because the CP-Scheduler may delay the execution of job *j*'s map tasks in order to execute the map tasks of a newly arriving job with an earlier deadline. One approach to solve this problem is to give execution priority to all of job *j*'s map tasks so that they can execute before other map tasks. Initially, this approach was used and implemented by adding constraints to the OPL Model that stated that map tasks should be scheduled to execute at their originally scheduled times and not be rescheduled. However, further testing showed that this solution is not ideal when it comes to minimizing the number of late jobs because jobs that have an earlier deadline may have to wait to be executed, leading to these jobs missing their deadlines. The solution that is used to avoid these problems is to prevent

the CP-Scheduler from assigning reduce tasks of a job *j* to TaskTrackers until all of job *j*'s map tasks have completed executing (as shown in line 29 of Algorithm 5.1). This approach guarantees that reduce tasks can complete their execution once they are scheduled on a TaskTracker, and it also allows the tasks of jobs with an earlier deadline to be executed first.

5.5 Performance Evaluation of the HCP-RM Technique

A rigorous performance evaluation of the HCP-RM technique (CP-Scheduler) is conducted on a Hadoop cluster deployed on Amazon EC2, which is a public cloud that provides Infrastructure-as-a-Service, to determine its effectiveness and to obtain insights into system behaviour and performance. The performance of the HCP-RM technique is compared to that of an Earliest Deadline First Hadoop scheduler (abbreviated EDF-Scheduler). The comparison with the EDF-Scheduler is made to investigate if the HCP-RM technique is more effective than the well-known EDF scheduling policy when matchmaking and scheduling an open stream of MapReduce jobs with deadlines. The EDF-Scheduler is implemented in Hadoop by defining a class called `EDF_Scheduler` in the package `org.apache.hadoop.mapred`. More specifically, the implementation of the `EDF_Scheduler` is based on the implementation of Hadoop's default FIFO scheduler, `JobQueueTaskScheduler` (discussed in Section 5.2.1) with changes that are made to the `JQ-JIPL` and `ETIL` classes. In the `ETIL` class the `resortInitQueue()` method is modified to sort the queue in non-decreasing order of job deadlines (i.e., priority is given to the jobs with an earlier deadline). Similarly, the `JQ-JIPL` class' `JobSchedulingInfoComparator` is also modified to sort jobs in non-decreasing order of their deadlines.

The rest of this section is organized as follows. The experimental setup and the performance metrics used to evaluate the HCP-RM technique and the EDF-Scheduler are described in Section 5.5.1. Next, descriptions of the two workloads that are used in the experiments are provided in Section 5.5.2 and Section 5.5.3.

5.5.1 Experimental Setup

The experiments conducted to evaluate the performance of the HCP-RM technique and the EDF-Scheduler are performed on a Hadoop cluster deployed on Amazon EC2. Amazon EC2 allows consumers to launch virtual machines (VMs) called *instances*. After launching these instances, consumers can connect to the instance to deploy and run their own applications. Amazon EC2 also provides various *instance types*, which are pre-configured VMs that have predetermined CPU, memory, storage, and networking capacity. The cost of running the instance depends on the type of instance deployed, and users are charged by the hour.

The Hadoop cluster (recall Section 2.4) deployed on Amazon EC2 contains 1 master node and 10 slave nodes with each slave node configured to have one map task slot and one reduce task slot each. The size of the Hadoop cluster is in line with the experimental platforms used by other researchers (see [71] and [72], for example). Each node in the cluster is an Amazon EC2 `m3.medium` instance, which is a fixed performance instance that provides a good balance of compute, memory, and network resources [108]. Each `m3.medium` instance runs Ubuntu 13.04 and is launched with a 2.5 GHz Intel Xeon E5-2670 v2 (Ivy Bridge) CPU and 3.75 GB of RAM.

For both the HCP-RM technique and the EDF-Scheduler, the following performance metrics are used to evaluate the performance of the respective systems:

- *Proportion of late jobs* (*P*) (recall Section 4.4.1)
- *Average job turnaround time* (*T*) (recall Section 4.4.1)
- *Average job matchmaking and scheduling time* (*O*): *O* is measured using Java's `System.nanoTime()` [102] method and is calculated as the total processing time required by the respective technique (i.e., HCP-RM technique or EDF-Scheduler) to perform matchmaking and scheduling in an experiment divided by the total number of jobs arriving on the system during the experiment.

To generate each of the values shown in the graphs presented in Section 5.6 and Section 5.7, the experiments are run long enough to ensure the system reached a steady state. Each experiment lasted about 24 hours and a subset of the experiments was run for a higher length of time. No significant change is observed in the trends in variation of a performance metric resulting from a variation of a given workload or system parameter. Thus, the 24-hour long experiments are deemed adequate to evaluate the relative performance between the HCP-RM technique and the EDF-Scheduler.

5.5.2 Hadoop WordCount Workload

The *Hadoop WordCount Workload* comprises an open stream of Hadoop WordCount jobs with deadlines (recall the discussion of a WordCount job presented in Section 2.3). WordCount is a benchmark that is commonly used to evaluate the performance of a Hadoop cluster and has been used by a number of other researchers (see [70], [72], and [73] for example). Furthermore, WordCount is representative of a class of real world MapReduce jobs that focus on extracting a small amount of interesting data from a large dataset [109]. The Hadoop WordCount Workload has three job types:

- *small*: processes 512 MB of data (8 text files of 64 MB each)

- *medium:* processes 1 GB of data (16 text files of 64 MB each)
- *large*: processes 2 GB of data (32 text files of 64 MB each)

The input data for each job type are e-books (in plain text format) that are obtained from Project Gutenberg (www.gutenberg.org). For each job type, the number of map tasks in the job is equal to the number of input files to process, and the number of reduce tasks is set to 9, which is in line with the recommended value stated in the Hadoop documentation [25]. Each map task requires processing 64MB of data, which is the default block size in HDFS [25].

A `JobSubmitter` is implemented using Java to submit an open stream of WordCount jobs at a specified arrival rate (λ) to the Hadoop cluster. In line with [70], the arrival of jobs to the system is generated using a Poisson process, and the values of the arrival rate (λ): 1/150, 1/140, 1/135, and 1/125 jobs per sec are chosen such that a moderate to high resource utilization between approximately 80% to 95% is generated on the Hadoop cluster described in Section 5.5.1. Each job type has an equal probability of being submitted to the system. For consistency, the same predetermined seed for the random number generator is used in the experiments for the HCP-RM technique and the EDF-Scheduler such that the same sequence of jobs with the same respective arrival times is generated. The other workload parameters are described next. The earliest start time of each job j (s_j) is set to its arrival time, and the deadline of each job j is calculated as: $d_j = s_j + SET_j^{max} * em$ where SET_j^{max} is the maximum execution time of the job j and em is the execution time multiplier. The purpose of em is to give the job some slack time, and it is generated using a uniform distribution within the interval [1, 3].

5.5.3 Hadoop Synthetic Workload

In addition to using the Hadoop WordCount Workload that captures the impact of both CPU and I/O characteristics of the application on performance, a synthetic workload, which is referred to as the *Hadoop Synthetic Workload*, is also used because it is flexible and enables the investigation of the impact of varying a given workload parameter on system performance. For example, using a synthetic workload, the effect of workload parameters such as the execution time of the jobs, which cannot be directly and accurately controlled using real workloads, can be investigated. The Hadoop Synthetic Workload is a variation of the Generic Synthetic MapReduce Workload (described in Section 4.4.3), and it is used to perform experiments on a real system such that the effect of various real system overheads on the behaviour and performance of the system, which are difficult to reproduce on simulation-based analyses, can be captured.

To perform experiments using the Hadoop Synthetic Workload, a new Hadoop application called *Simulate Execution Time* (abbreviated SimExec) is devised. The map and reduce tasks of SimExec jobs occupy the map/reduce task slots of the TaskTrackers that they are assigned to execute on by sleeping for a specified amount of time (invoking Java's `Thread.sleep()` method [110]). The time that the tasks sleep for is used to simulate the amount of time required for processing. Note that while the task is sleeping, no other task can run on the task slot where a task has been assigned, until the task wakes up from its sleep and informs the JobTracker that it has completed executing.

Table 5.1 summarizes the parameters for generating the Hadoop Synthetic Workload. A Poisson process is used to generate the arrival of jobs at a specified rate (λ), which is in line with [70]. The values of λ are chosen to subject the systems (HCP-RM and

the EDF-Scheduler) to moderate to high system load, leading to an average resource utilization of between approximately 80% to 90% on the Hadoop cluster described in Section 5.5.1. In these experiments the systems are not subject to a low system load (resulting from low arrival rates). This is because a comparison of the effectiveness of the resource management algorithms when the system load is low is uninteresting since a low load generates a low contention for resources and is observed to lead to both algorithms achieving a value of 0 for *P*. Thus, comparing the performance of the systems when the system load is low was deemed inappropriate in evaluating the effectiveness of the resource management algorithms.

The attributes of each job *j* that arrives on the system are generated as follows. The earliest start time of a job *j* (s_j) is set to the arrival time of the job (at_j), which is in line with Hadoop 1.2.1's schedulers [25]. As shown in Table 5.1, the number of map tasks (k_j^{mp}), the number of reduce tasks (k_j^{rd}), as well as the map task execution times (*me*) and reduce task execution times (*re*) are generated using discrete uniform (*DU*) distributions similar to the Generic Synthetic MapReduce workload. Note that for SimExec jobs, *me* and *re* represent the time that a map task or a reduce task occupies the map task slot or reduce task slot of a TaskTracker. During preliminary experiments, it was discovered that a TaskTracker requires approximately 5 sec for initialization before the user-defined code of the map task or the reduce task is executed. Thus, the amount of time that a map task of a SimExec job needs to sleep for is calculated as the difference between *me* and the task setup time. Similarly, the amount of time that a reduce task needs to sleep for is calculated as the different between *re* and the task setup time. The sleep time of the tasks need to be compensated for the task setup time so that the task occupies a task slot of a TaskTracker

for the time specified by *me* or *re*. Similar to the Generic Synthetic MapReduce workload, a job *j*'s deadline (d_j) is calculated as the sum of s_j and the product of SET_j^R (which is the execution time when job *j* executes at its maximum degree of parallelism on *R*) and an execution time multiplier (*em*). The parameter *em* is generated using a uniform distribution (*U*) as follows: $U(1, em_{max})$ where em_{max} is the upper bound of *em*.

Table 5.1. System and Workload Parameters for the Hadoop Synthetic Workload.

Parameter	***Values***	***Default Value***
Job		
Arrival rate, λ (jobs/sec)	$\lambda = \{1/32.5, 1/30, 1/27.5\}$	$\lambda = 1/30$
Earliest start time, s_j (sec)	$s_j = at_j$	-
No. of Map Tasks, k_j^{mp}	$k_j^{mp} \sim DU(1, 10)$	-
No. of Reduce Tasks, k_j^{rd}	$k_j^{rd} \sim DU(1, k_j^{mp})$	-
Deadline, d_j (sec)	$d_j = \lceil s_j + SET_j^R * em \rceil$ where $em \sim U(1, em_{max})$ and $em_{max} = \{20, 25, 30\}$	$em_{max} = 25$
Task		
Map task execution time, me (sec)	$me \sim DU(1, me_{max})$ where $me_{max} = \{15, 20, 25\}$	$me_{max} = 20$
Reduce task execution time, re (sec)	$re = \left\lceil (3 * \sum_{t \in T_j^{mp}} e_t)/k_j^{rd} \right\rceil + DU(1,10)$	-

Note: DU = discrete uniform distribution, *U* = uniform distribution

The distributions used to generate k_j^{mp}, k_j^{rd}, *me*, and *re* are adopted from [53], whereas d_j (a parameter that is not used in [53]) is generated based on [70]. In addition, the use of a Poisson process to generate job arrivals is in line with [70]. The values used in the distributions are different from those described in Section 4.4.3 because they are adjusted to keep in line with the smaller number of resources used in the Hadoop cluster compared to that used in the simulation experiments described in Section 4.6. More specifically, the values for λ are chosen to generate high resource utilization; however, since there are only 10 TaskTrackers (resources) in the Hadoop cluster, which is smaller than the 50 resources

used in the simulated system, me_{max} is reduced and em_{max} is increased to generate a reasonable workload. For similar reasons, the max value of the *DU* distribution that is used for generating the number of map tasks is set to 10 instead of 100.

Factor-at-a-time experiments, where one parameter is varied and the other parameters are kept at their default values, as shown in the "Default Value" column of Table 5.1, are conducted to study the effect of the various workload parameters. The results of these experiments are discussed in Section 5.6.1.

5.6 Results of the Performance Evaluation

This section discusses the results of the experiments conducted to compare the performance of the HCP-RM technique (referred to simply as HCP-RM) with that of the EDF-Scheduler (abbreviated EDFS) when using the Hadoop WordCount Workload (see Section 5.6.1) and the Hadoop Synthetic Workload (refer to Section 5.6.2). Note that in the figures that show the results of *T* and *O* (see Figure 5.8, for example), *T* is displayed as a bar graph that uses the scale on the left Y-axis and *O* is displayed as a sequence of points that uses the scale on the right Y-axis.

5.6.1 Results of Experiments Using the Hadoop WordCount Workload

Figure 5.5 and Figure 5.6 compare the performance of HCP-RM with that of EDFS when using the Hadoop WordCount Workload (described in Section 5.5.2). As shown in the figures, HCP-RM outperforms EDFS by a large margin in terms of *P* (up to 94% and on average 72%) and *T* (up to 65% and on average 52%). This is a result of HCP-RM being able to effectively interleave the execution of multiple jobs and efficiently make use of the system's resources such that the number of jobs that miss their deadlines is minimized. The poor performance of EDFS with regards to *P* and *T* is attributed to its focus on only

mapping the job with the earliest deadline and not interleaving the execution of multiple jobs. When the jobs have long execution times, not interleaving the execution of multiple jobs tends to lead to more late jobs because it is possible for jobs with closer deadlines to arrive on the system during the execution of another job with a later deadline. Thus, the results demonstrate that EDFS cannot effectively handle an open stream of job arrivals, and it may be more effective in matchmaking and scheduling a fixed number of jobs (e.g., a batch workload) because all the jobs in the workload are known ahead of time. Overall, the results demonstrate that HCP-RM can effectively match make and schedule an open stream of MapReduce jobs with deadlines.

Figure 5.5. HCP-RM vs EDFS: effect of λ on *P* when using the Hadoop WordCount Workload.

Figure 5.6 (refer to sequence of points) shows that EDFS achieves a lower *O* (on average 0.03 sec.) compared to HCP-RM (on average 9.04 sec). HCP-RM has a higher *O* because it uses a more complex matchmaking and scheduling algorithm that requires

generating and solving a constraint program using IBM CPLEX. Conversely, EDFS simply maintains a job queue that sorts jobs in non-decreasing order of their respective deadlines and chooses the first job in its queue to map. Thus, the *O* of EDFS tends to remain relatively stable even when λ increases. However, it is observed that HCP-RM's *O* tends to increase with λ. The reason for this is because when λ increases, jobs arrive on the system more frequently, leading to a higher contention for resources. This in turn causes HCP-RM to have more jobs to map (and more decision variables and constraints to process) each time it needs to generate and solve an OPL Model. Although HCP-RM's *O* is higher compared to that of EDFS, *O*/*T*, which is an indication of the processing overhead in relation to the average job turnaround time, is still very low (less than 0.64%). This means that the *O* of HCP-RM is acceptable for the given workload.

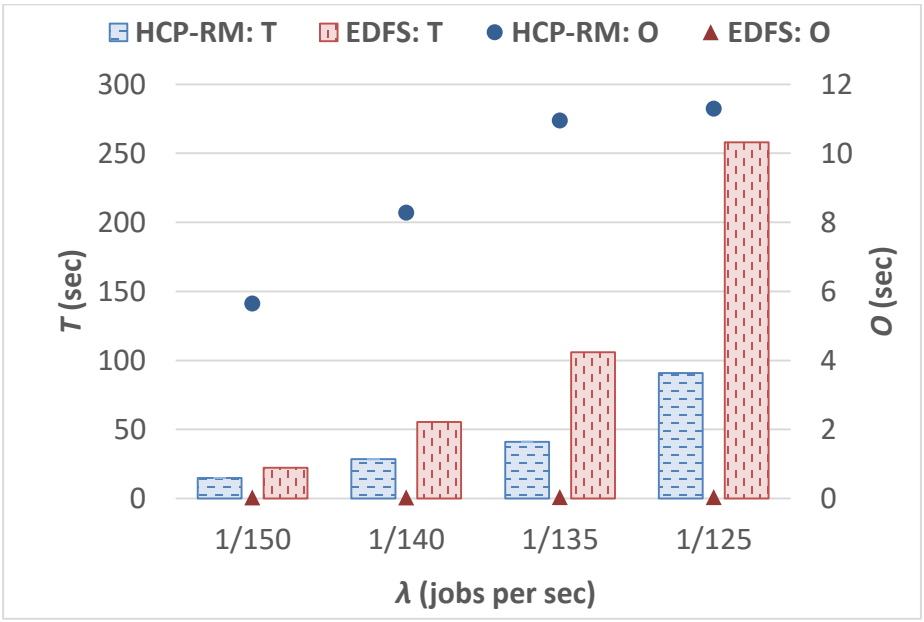

Figure 5.6. HCP-RM vs EDFS: effect of λ on *T* and *O* when using the Hadoop WordCount Workload.

5.6.2 Results of Experiments Using the Hadoop Synthetic Workload

The results of the experiments using the Hadoop Synthetic Workload are presented and discussed in this section. The effects of three workload parameters on system performance are investigated: job arrival rate, task execution times, and job deadline.

5.6.2.1 Effect of Job Arrival Rate

As shown in Figure 5.7 and Figure 5.8, for both systems, P and T increase with λ because at high values of λ there is a high contention for resources and not all the jobs are able to start executing at their earliest start times. However, it is observed that HCP-RM outperforms EDFS in terms of both P and T for all the values of λ experimented with. More specifically, HCP-RM is observed to achieve a P and O that is on average 77% lower and 65% lower, respectively, compared to the P and O achieved by EDFS. However, the O (refer to the sequence of points in Figure 5.8) of HCP-RM is observed to be on average 3.4 sec, which is much higher compared to the O achieved by EDFS (9 ms on average). As described in Section 5.6.1, HCP-RM's O increases with λ due to jobs arriving on the system at a faster rate, which leads to the CP Optimizer having to solve an OPL Model with more decision variables and constraints. In comparison to EDFS, HCP-RM puts more effort into deciding how to map jobs onto resources to minimize P, leading to a higher O. The benefits of this are captured in the superior performance demonstrated by HCP-RM with its lower P and T, while still maintaining a small O/T (0.87% on average over all the values of λ experimented with).

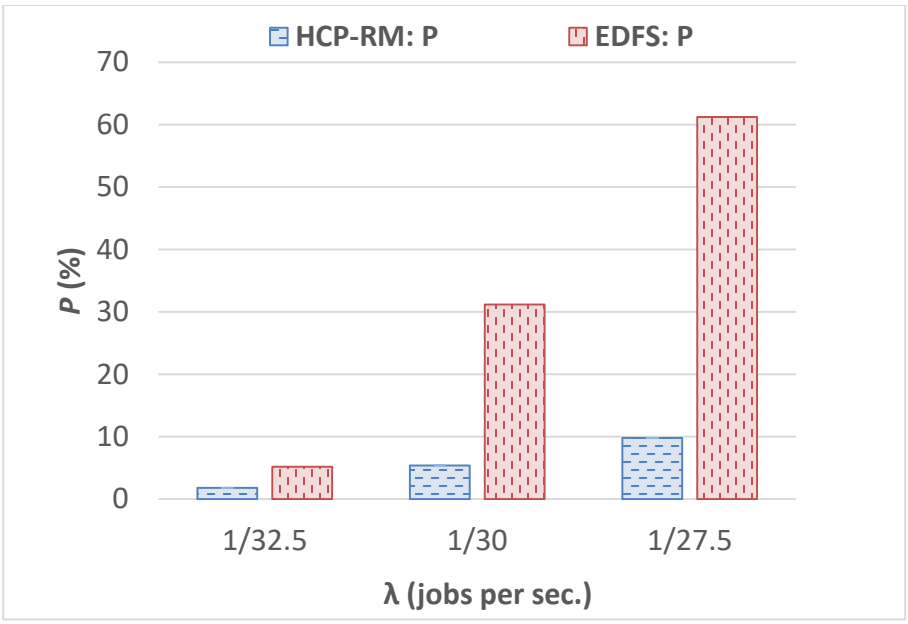

Figure 5.7. HCP-RM vs EDFS: effect of λ on *P* when using the Hadoop Synthetic Workload.

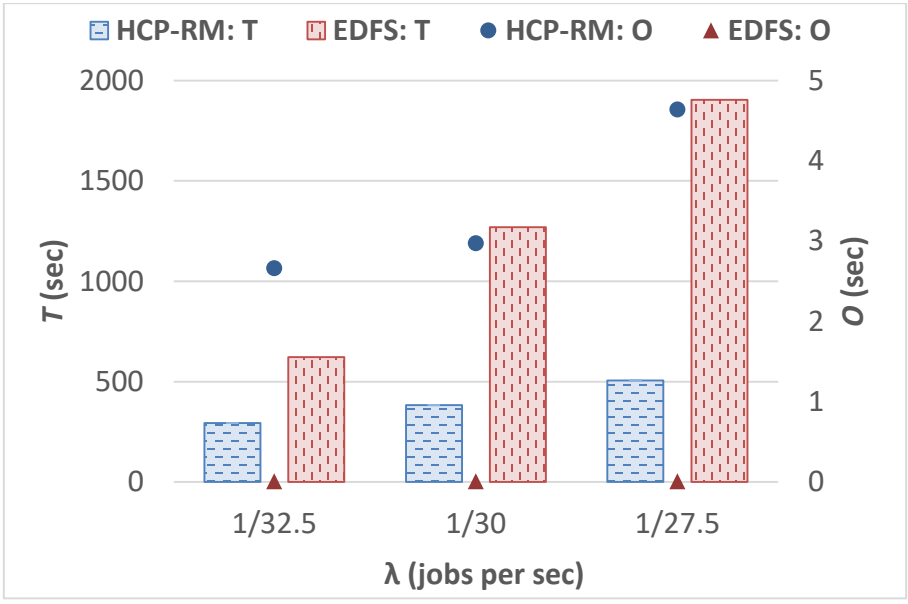

Figure 5.8. HCP-RM vs EDFS: effect of λ on *T* and *O* when using the Hadoop Synthetic Workload.

5.6.2.2 *Effect of Task Execution Times*

As expected, when me_{max} is increased, both P and T increase for both systems (refer to Figure 5.9 and Figure 5.10) due to jobs executing on the resources for a longer period of time, which in turn leads to a high contention for resources. Moreover, it is observed that HCP-RM outperforms EDFS in terms of both P and T when me_{max} is equal to 20 or 25 sec. However, when me_{max} is small (equal to 15 sec) EDFS achieves a slightly lower P compared to HCP-RM (4.0% vs 4.4%). The slightly inferior performance of HCP-RM in this case can be attributed to its high O having a larger impact on jobs with small execution times, resulting in these jobs missing their deadlines.

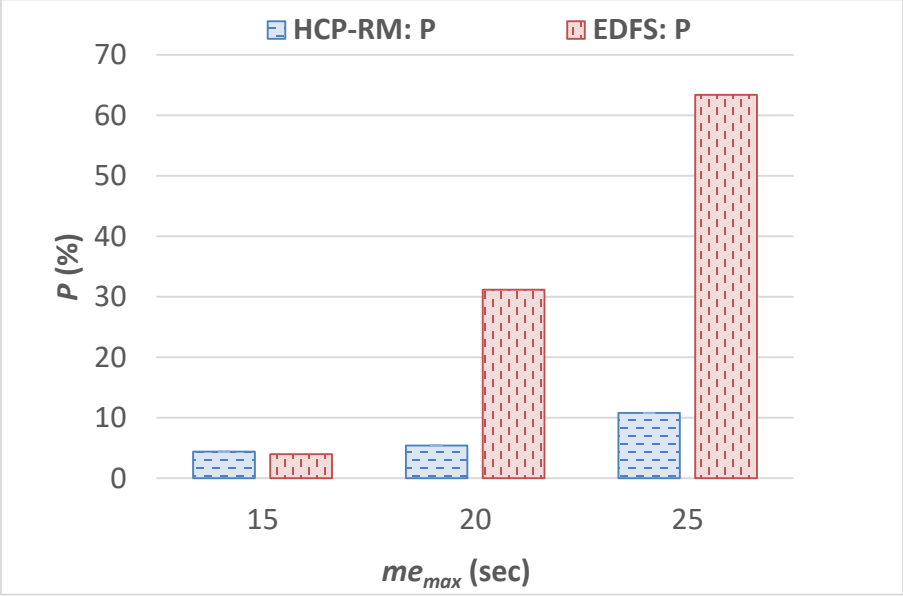

Figure 5.9. HCP-RM vs EDFS: effect of me_{max} on P when using the Hadoop Synthetic Workload.

With regards to O (refer to the sequence of points in Figure 5.10), the relative performance achieved by HCP-RM and EDFS is similar to that observed in the results discussed earlier: the O achieved by HCP-RM is higher compared to the O achieved by

EDFS, but the processing overhead as indicated by O/T, is still small (less than 0.8%). The reason for the HCP-RM's O increasing with me_{max} can be attributed to jobs remaining in the system for a longer period of time and potentially overlapping with the execution of a higher number of jobs. This in turn increases the OPL Model generation and solving times due to the higher number of constraints and decision variables that need to be processed.

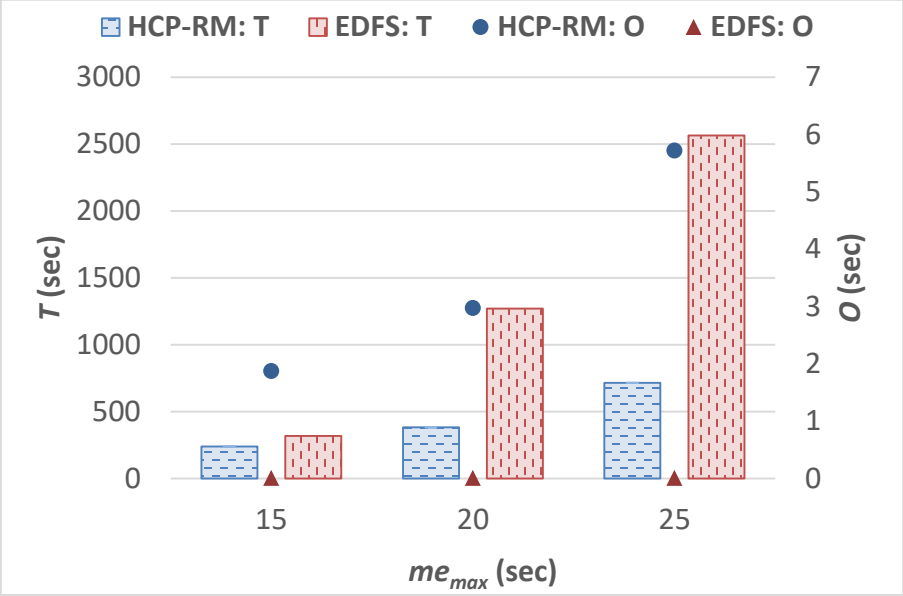

Figure 5.10. HCP-RM vs EDFS: effect of me_{max} on T and O when using the Hadoop Synthetic Workload.

5.6.2.3 Effect of Job Deadlines

For both systems, Figure 5.11 and Figure 5.12 demonstrate that P decreases while T tends to be stable as em_{max} increases. The superior performance of HCP-RM over that of EDFS is demonstrated once again when em_{max} is varied. More specifically, it is observed that HCP-RM outperforms EDFS in terms of both P (on average 58% lower) and T (on average 65% lower). However, when considering O (refer to the sequence of points in Figure 5.12), it is observed that EDFS achieves an O of 9 ms on average and outperforms HCP-RM, which has an O of 2.92 sec on average. From Figure 5.12 (refer to the sequence

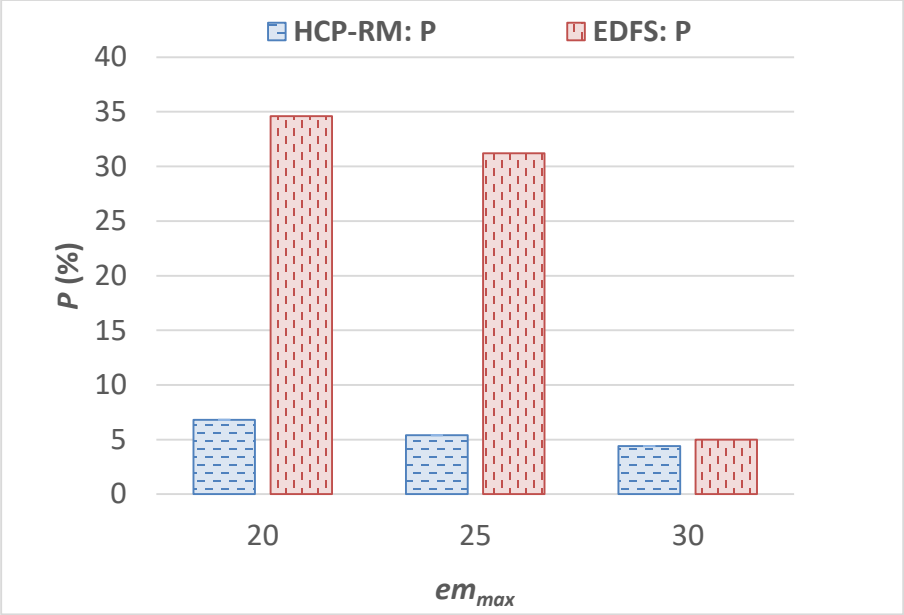

Figure 5.11. HCP-RM vs EDFS: effect of em_{max} on P when using the Hadoop Synthetic Workload.

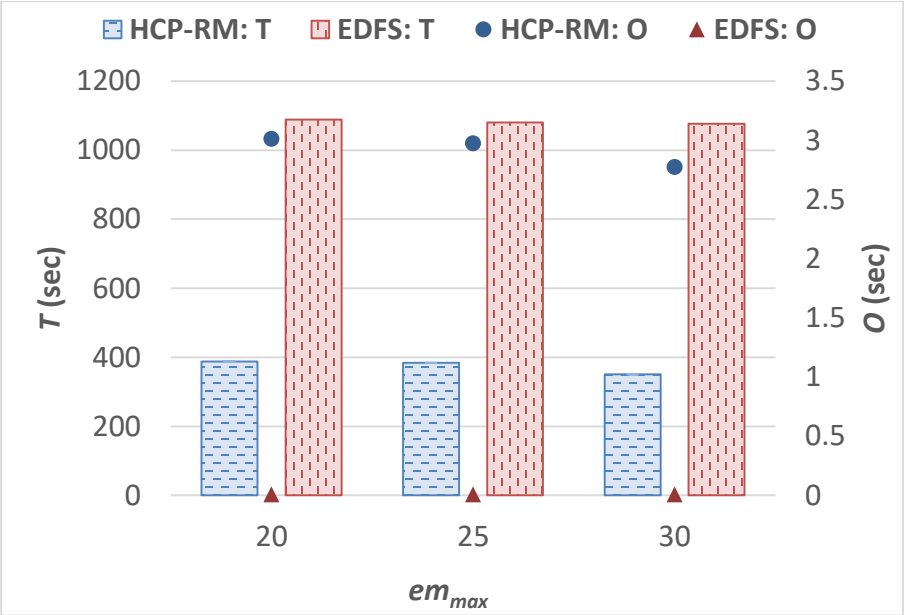

Figure 5.12. HCP-RM vs EDFS: effect of em_{max} on T and O when using the Hadoop Synthetic Workload.

of points), it is also observed that the HCP-RM's O increases as em_{max} decreases. This is because at low values of em_{max} jobs have tighter deadlines, and thus HCP-RM requires

more time to generate a schedule that minimizes the number of late jobs. Note that the higher O of the HCP-RM is tolerable for the workload experimented with, as indicated by the small value of O/T: on average 0.78% over all the values of em_{max} experimented with.

5.7 Investigation of Error in User-estimated Execution Times

As described in Section 2.6, previous studies on real systems have shown that user estimates of execution times are often error prone or inaccurate [16][17][18]. For example, in [77], the authors analyzed a workload trace and showed that approximately 98% of jobs have their execution times overestimated and 2% of jobs have their execution times underestimated. In this section, the results of an investigation into how the error or inaccuracy in the user-estimated execution times, submitted as part of the job's SLA, can affect the performance of HCP-RM in terms of P, T, and O are presented and discussed.

Note that only the performance of HCP-RM is investigated because the EDF-Scheduler does not use user-estimated execution times in its matchmaking and scheduling algorithm. The system used in this investigation is the 11-node Hadoop cluster deployed on Amazon EC2 as described in Section 5.5.1. In addition, the workload used in this investigation is the Hadoop Synthetic Workload (described in Section 5.5.3) with the default parameter values. However, to investigate the impact of error at different levels of load on the system, λ is varied from 1/60 jobs per sec to 1/25 jobs per sec to generate low (~45%), moderate (~80), and high (~95%) system load. The Hadoop Synthetic Workload is used instead of the Hadoop WordCount Workload because in the synthetic workload the task execution times can be systematically and accurately controlled.

The rest of this section is organized as follows. First, in Section 5.7.1, the description of the models used to generate error for the user-estimated execution times are

described. A discussion of how the error in user-estimated execution times can affect the laxity (or slack time) of jobs is provided in Section 5.7.2. Lastly, the results of the experiments conducted to investigate how error in user-estimated execution times can affect the performance of HCP-RM are presented in Section 5.7.3 and Section 5.7.4.

5.7.1 Models for Generating Error in User-estimated Execution Times

To generate the error in user-estimated task execution times, two models are used: (1) Constant Error Model and (2) Feitelson's Error Model. When using the *Constant Error Model* [111], all the jobs submitted to the system have their user-estimated task execution times overestimated or underestimated by a constant percentage of their actual runtimes. More specifically, the *estimated execution time* of a task t (e_t^{est}) is calculated as follows:

$$e_t^{est} = e_t \times (1 + f) \tag{5.1}$$

where e_t is the *actual runtime* of task t and f is the *execution time error factor*. If the value of f is greater than 0, it means that the estimated task execution times are greater than the actual runtimes of the task (i.e., task execution times are *overestimated*). On the other hand, if f is less than 0, it means that the estimated task execution times are less than the actual runtime of the tasks (i.e., task execution times are *underestimated*). For example, if f is equal to 0.1, the estimated task execution times are overestimated by 10%. On the other hand, if f is equal to -0.05, the estimated task execution times are underestimated by 5%. Note that f cannot be less than or equal to -1; otherwise, e_t^{est} will be less than or equal to 0.

Feitelson's Error Model [112] is based on the analysis of real workload traces collected from various sources including Cornell Theory Center, Swedish Royal Institute of Technology, and the San Diego Supercomputer Center. The algorithm used to generate the estimated task execution times using Feitelson's Error Model is summarized in

Algorithm 5.4. A walkthrough of the algorithm is provided next. With a probability of 10%, the user-estimated task execution time is very accurate and is calculated as: e_t^{est} = 0.99*e_t (lines 1-3). On the other hand, with a probability of 90%, e_t^{est} is not accurate (line 4) and is generated as follows. If the supplied task execution time, e_t, is less than a *task execution time threshold* ($e_t^{threshold}$) then e_t^{est} is calculated as 10*e_t (see lines 5-6), meaning that the task execution time is highly overestimated. Otherwise, e_t is greater than or equal to $e_t^{threshold}$ and e_t^{est} is calculated as e_t/u where u is an uniformly distributed variable from (0, 1] (lines 7-9). This means that the smaller the value of u, the higher the value of e_t^{est} that is generated, resulting in highly overestimated task execution times. The closer that u is to 1, the closer e_t^{est} is to e_t, meaning that the estimated task execution time is more accurate.

Algorithm 5.4: Feitelson's Error Model

Input: e_t, actual task run time (in sec)
Output: e_t^{est}, estimated task execution time (in sec)

```
1:  rv ← Generate a uniformly distributed random variable from [0, 1].
2:  if rv <= 0.1 then
3:      return 0.99 * e_t
4:  else
5:      if e_t < e_t^threshold then
6:          return 10 * e_t
7:      else
8:          u ← Generate a uniformly distributed random variable from (0, 1].
9:          return e_t / u
10:     end if
11: end if
```

The default value of $e_t^{threshold}$ is 90 sec; however, for the Hadoop Synthetic Workload (recall Section 5.5.3) that this investigation uses, it was found that when $e_t^{threshold}$ is kept at its default value, all the map tasks and a majority of reduce tasks had their estimated task execution times calculated using line 6: $e_t^{est} = e_t * 10$. This skewed the

estimated task execution times towards e_t *10. To make sure that the task execution times are not skewed, the $e_t^{threshold}$ is set to 20 sec, which is the default value for generating the map task execution times (refer to Table 5.1). This allowed e_t^{est} to be calculated using line 6 and line 9 approximately the same number of times.

5.7.2 Laxity of Jobs in the Presence of Error in User-estimated Execution Times

In this investigation, the deadline of each job submitted to the system is calculated using the respective job's estimated task execution times, which may be inaccurate, and not the actual runtime of the tasks as generated by the synthetic workload generator. The rationale behind this decision is that a user who submits a job will generate a deadline for their job using the estimated task execution times that they supply. This means that if the deadline of a job is calculated using *overestimated* task execution times, the job will have *more* laxity (or slack time) compared to when there is no error in the user-estimated task execution times. Alternatively, if the deadline of a job is calculated using *underestimated* task execution times, the job will have *less* laxity compared to when there is no error in user-estimated task execution times. Recall from Section 3.1.1 that the laxity of a job j is the extra time job j has to complete its execution before its deadline and is calculated as follows: $L_j = d_j - s_j - SET_j$ where d_j is the deadline of job j, s_j is the earliest start time of job j, and SET_j is the sample execution time of job j.

The equations that are described next are based on the Constant Error Model and are helpful when analyzing and understanding the results of the experiments presented in Section 5.7.3. The insights gained from this discussion can also be applied to the results of the experiments using Feitelson's Error Model, described in Section 5.7.4.

The *estimated laxity* of a job *j* (L_j^{est}), which is the laxity that HCP-RM expects a job as having when it is submitted to the system for execution, is calculated as follows:

$$\begin{aligned} L_j^{est} &= d_j - s_j - SET_j \times (1 + f) \\ &= (s_j + SET_j \times (1 + f) \times em) - s_j - SET_j \times (1 + f) \\ &= SET_j \times (1 + f) \times [em - 1] \end{aligned} \quad (5.2)$$

where d_j is the deadline of job *j*, s_j is the earliest start time of job *j*, SET_j is the sample execution time of job *j* (calculated without error) (see Section 3.1.1), *f* is the execution time error factor, and *em* is the execution time multiplier (see Section 5.5.3). As described earlier in this sub-section, the deadline of each job submitted to the system is calculated using the respective job's estimated task execution times, which may contain error, and thus the deadline of a job *j* (d_j) is calculated as $s_j + SET_j * (1 + f) * em$. If *f* is equal to 0, meaning that there is no error in task execution times, Eq. 5.2 is the same as Eq. 3.1 (described in Section 3.1.1).

The *actual laxity* of a job *j* (L_j^{act}), which is the laxity calculated using the actual run time of the job (i.e., the laxity that the job has in reality), is calculated as follows:

$$\begin{aligned} L_j^{act} &= d_j - s_j - SET_j \\ &= (s_j + SET_j \times (1 + f) \times em) - s_j - SET_j \\ &= SET_j [(1 + f) em - 1] \end{aligned} \quad (5.3)$$

The main difference between L_j^{act} and L_j^{est} is that for L_j^{est} (see Eq. 5.2) the sample job execution time (SET_j) is multiplied with $(1 + f)$, whereas for L_j^{act} (see Eq. 5.3) the sample job execution time is not multiplied with $(1 + f)$ and thus does not contain error.

5.7.3 Results of Experiments Using the Constant Error Model

Figure 5.13 presents the effect of *f* on *P* at different arrival rates. When the task execution times are underestimated (i.e., $f < 0$), *P* increases, and when the task execution times are overestimated (i.e., $f > 0$), *P* decreases. The reason for *P* decreasing as *f* increases

is due to jobs having higher values of L_j^{act} when f increases, which means jobs have more time to complete their execution before their respective deadlines (i.e., jobs have less stringent deadlines). The high values of P when f is less than or equal to -0.5 can be attributed to jobs having very small values of L_j^{act} (i.e., jobs have very stringent deadlines). Furthermore, when f is very small (e.g., f is -0.9), it is possible that jobs have deadlines that cannot be satisfied because they were calculated based on severely underestimated execution times, and thus there is not enough time for the job to finish executing before its deadline (e.g., the job may have an L_j^{act} of less than 0).

Figure 5.13. Constant Error Model: effect of f on P.

As shown in Figure 5.14, for a given arrival rate, the general trend in performance that is observed is that T increases as f decreases. This trend is especially evident when λ is 1/25 jobs per sec. The high values of T when f is small (e.g., less than 0) can be attributed to jobs having less laxity and higher execution times than what HCP-RM expects, which means that the jobs have more stringent deadlines than what HCP-RM is expecting. Thus,

more jobs miss their deadlines and P increases (refer to Figure 5.13 when f is less than 0). When a job misses its deadline, HCP-RM delays executing the remaining tasks of the job that have not started executing in favour of executing newly arriving jobs, which have not missed their deadlines. This is done to lower P, but also leads to an increase in T.

Figure 5.14. Constant Error Model: effect of f on T.

Figure 5.15 illustrates the effect of f on the O of HCP-RM. For a given arrival rate, when the task execution times are underestimated ($f < 0$), it is observed that O decreases and when the task execution times are overestimated ($f > 0$), it is observed that O increases. Overall, the general trend is that O increases as f increases. This is because at higher values of f, HCP-RM expects that jobs submitted to the system have higher execution times compared to when f is small, resulting in HCP-RM perceiving the system having a high contention for resources. This in turn leads to HCP-RM requiring more time to perform matchmaking and scheduling to ensure that P is minimized. Along with the increase in job execution times, the increase in f also causes HCP-RM to perceive jobs as having high

values of L_j^{est} (recall Eq. 5.2). For example, when f is -0.5 the average L_j^{est} of all the jobs submitted to the system is equal to 580 sec compared to 1740 sec when f is 0.5. The higher values of L_j^{est} give HCP-RM more options to explore scheduling different combination of tasks to ensure that the number of late jobs is minimized, which results in O increasing.

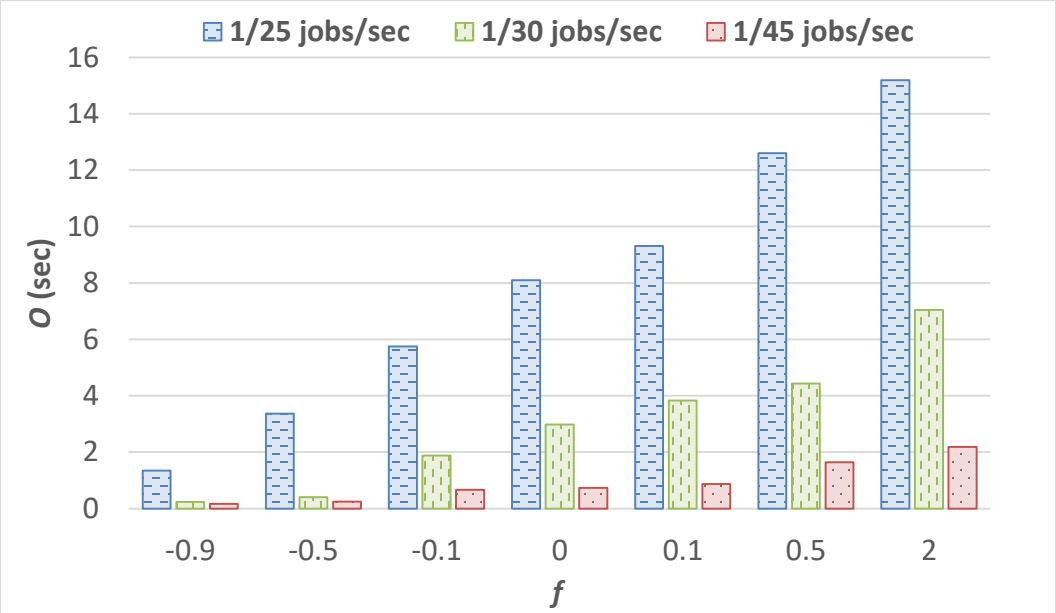

Figure 5.15. Constant Error Model: effect of f on O.

5.7.4 Results of Experiments Using Feitelson's Error Model

Figure 5.16, Figure 5.17, and Figure 5.18 present the results of the experiments in terms of P, T, and O, respectively, when using Feitelson's Error Model compared to the case where there is no error in the execution times of the jobs (referred to as the "No Error" case). As shown in Figure 5.16, for all the values of λ experimented with, P is much lower when using Feitelson's Error Model compared to the No Error case. This can be attributed to the fact that Feitelson's Error Model generates user-estimated task execution times that are highly overestimated. When the user-estimated execution times are overestimated, the values of L_j^{act} (recall Eq. 5.3) increase, which means that jobs have less stringent deadlines.

With regards to *T* (refer to Figure 5.17), it is observed that for all the values of λ experimented with, the results when using Feitelson's Error Model are higher compared to the results when using the No Error case. This can be attributed to HCP-RM focusing on generating a schedule that minimizes *P* and does not consider finding a solution to minimize *T*. On the other hand, in the No Error case, HCP-RM needs to schedule jobs to execute closer to their earliest start times because jobs have less slack time and more stringent deadlines, resulting in a low *T*. From Figure 5.18, it is observed that the *O* measured for the No Error case is a higher compared to the *O* measured when using Feitelson's Error Model. This can be attributed to the jobs in the No Error case having less laxity and more stringent deadlines, leading to HCP-RM requiring more time to generate a schedule to minimize the number of late jobs.

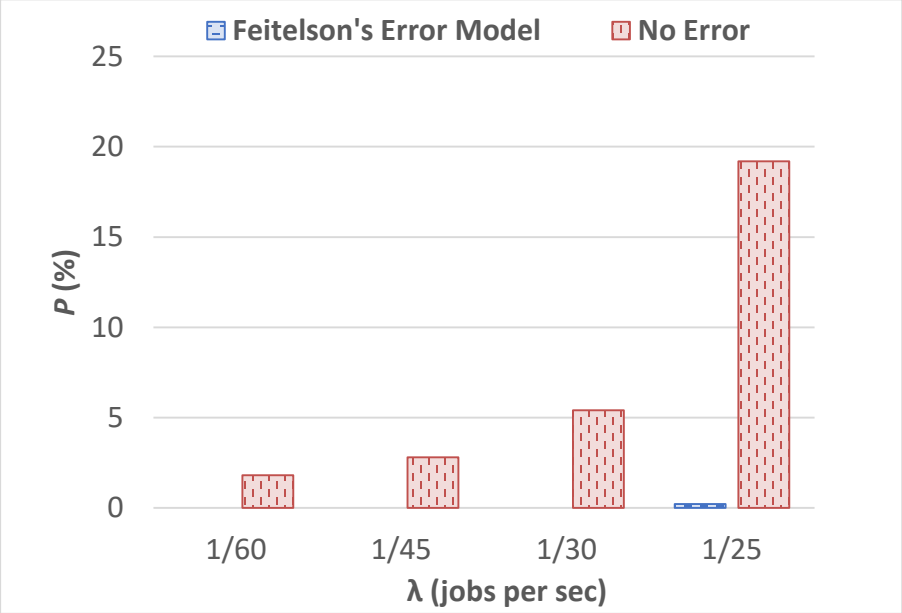

Figure 5.16. Feitelson's Error Model vs No Error: effect of λ on *P*.

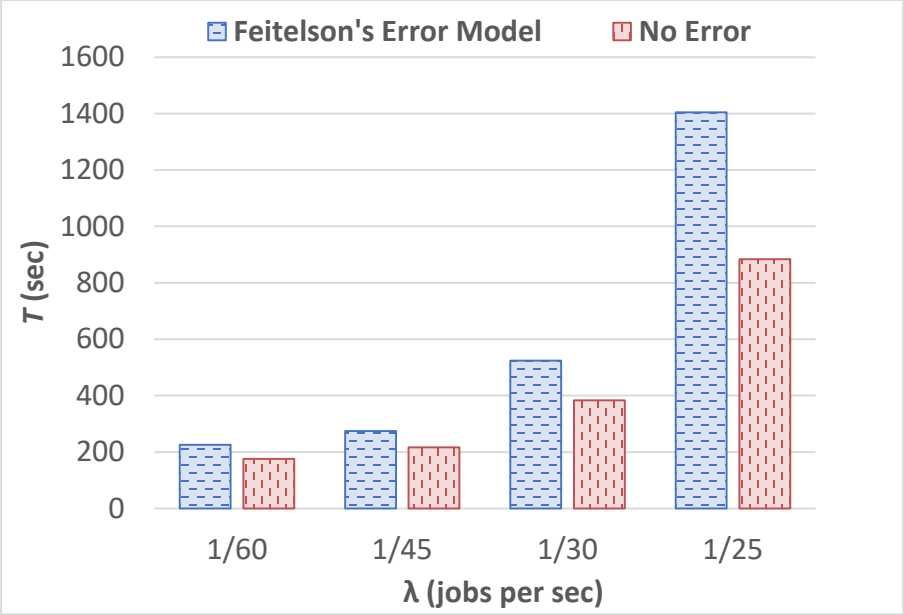

Figure 5.17. Feitelson's Error Model vs No Error: effect of λ on *T*.

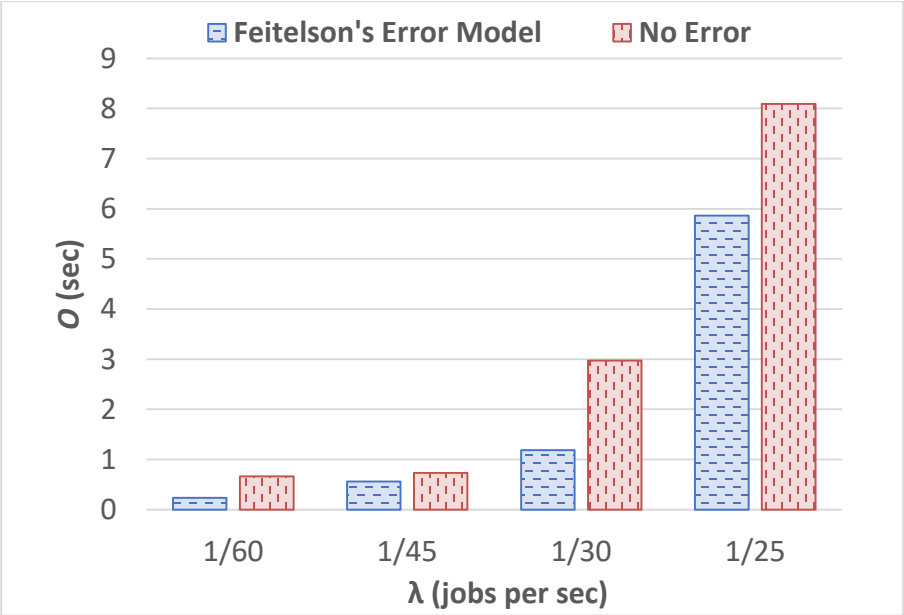

Figure 5.18. Feitelson's Error Model vs No Error: effect of λ on *O*.

5.8 Summary and Discussion

In this chapter, a data-locality-aware Hadoop Constraint Programming based Resource Management (HCP-RM) technique is presented. The HCP-RM algorithm is implemented in a new scheduler for Hadoop, called the CP-Scheduler. The objective of the CP-Scheduler (HCP-RM technique) is to minimize the number of late jobs when processing an open stream of Hadoop jobs with deadlines. This is accomplished by formulating the matchmaking and scheduling problem as an optimization problem using constraint programming and solving the problem using IBM CPLEX's CP Optimizer solving engine. A comprehensive performance evaluation of the HCP-RM technique is conducted on a Hadoop cluster deployed on Amazon EC2. The results of the performance evaluation demonstrate the effectiveness of the HCP-RM technique in being able to generate a schedule that leads to a small proportion of jobs missing their deadlines. In addition, experiments are performed to investigate how error in user-estimated execution times can affect system performance. The key insights into system behaviour and the inferences derived are summarized next.

- *Superiority of the HCP-RM technique*: The HCP-RM technique generates a schedule that leads to a lower or equal P (on average 60% lower) and a lower T (on average 59% lower) compared to the EDF-Scheduler over all the experiments performed. The highest performance improvement of the HCP-RM technique over the EDF-Scheduler tends to occur when the contention for resources is high (e.g., high λ, or high me_{max}, or small em_{max}).
    - The EDF-Scheduler's simple matchmaking and scheduling algorithm is not as effective in matchmaking and scheduling an open stream of MapReduce

jobs with deadlines. The EDF-Scheduler seems to be more suited for use in a closed system subject to batch workloads with a fixed number of jobs, where the set of jobs to execute are known ahead of time.

- *Efficiency of the HCP-RM technique*: The efficiency of the HCP-RM technique is demonstrated through the results of experiments described in Section 5.6. Although the *O* achieved by the HCP-RM technique is higher compared to the *O* achieved by the EDF-Scheduler, its superiority comes from being able to generate a schedule that leads to a small *P* and a small *T*. In all the experiments conducted, the *P* achieved by the HCP-RM technique is significantly lower than the *P* achieved by the EDF-Scheduler, while *O/T*, which is an indicator of resource management overhead, remains below 0.92% even when the resource utilization is high.
- *Effect of error in execution times:* The investigation of error in user-estimated execution times revealed that overestimation of execution times (which occurs more often than underestimation [16][77]) leads to a lower *P*, similar *T*, and slightly higher *O* compared to when there is no error. Conversely, underestimation of execution times leads to higher values of *P* and *T*, but gives rise to a lower *O* compared to the case when there is no error. Thus, it is more favourable for the system to have jobs with overestimated execution times compared to jobs with underestimated execution times.

Overall, the results obtained from prototyping and measurements made on a Hadoop cluster lead to the conclusion that the HCP-RM technique is an effective and efficient matchmaking and scheduling technique for processing MapReduce jobs with

deadlines on a Hadoop cluster. The HCP-RM technique achieves a small P and T, and an acceptable O/T over a wide range of workload and system parameters used in the experiments. Moreover, when there is error in execution times, it is observed that the HCP-RM technique maintained acceptable values of P when the execution times are slightly underestimated (e.g., f is equal to -0.1) or overestimated (f is greater than 0).

Chapter 6 Techniques for Handling Error/Inaccuracy in User-estimated Execution Times

The focus of this chapter is on describing techniques for handling inaccuracy or error in user estimates of job execution times (submitted as part of the job's SLA). The objective is to improve the robustness of the MRCP-RM and HCP-RM algorithms, described in Chapter 4 and Chapter 5, respectively. With little existing work on the handling of inaccuracies in user estimates of job runtimes in the context of MapReduce/Hadoop systems, the proposed techniques and experimental results presented can lead to new insights for users and system builders and make a strong contribution to the state of the art. None of the techniques described in Section 2.6 that handle errors/inaccuracies in user-estimated job execution times concern processing an open stream of MapReduce jobs with SLAs.

As described in Section 2.6, previous studies on real systems have shown that user estimates of job runtimes are often error prone/inaccurate and users often overestimate the execution times of their jobs [16][17][18]. Since the user-estimated execution times are used by the system to perform resource allocation and scheduling, error/inaccuracies in the execution times can hinder the ability of the resource management techniques from making effective scheduling decisions, leading to a degradation in system performance. For instance, an overestimated job execution time (i.e., a user requests more time than the job needs) causes resources to remain idle after a job completes before its estimated completion time, resulting in a low resource utilization. On the other hand, an underestimated job execution time (i.e., a user requests less time than the job needs) can cause jobs to be aborted prematurely due to the resource executing the job having to execute another job

that was already scheduled to start executing after the first job's expected completion time. This may result in the first job missing its deadline. Furthermore, if aborted, this incomplete job leads to a lower useful utilization of the system because the system's resources are being wasted executing a job that is aborted.

Moreover, the results of the investigation into how error in user-estimated execution times affect the performance of the HCP-RM technique (described in Section 5.7) showed that system performance in terms of *P*, *T*, and *O* is affected and the error in user-estimated execution times can also influence the matchmaking and scheduling decisions that the HCP-RM algorithm makes. For example, if a job has an actual runtime equal to 5 sec, but the user-estimated execution time is 10 sec (i.e., the job has an overestimated execution time), the resource management algorithm will only schedule the job to execute on a resource where there is an available time interval equal to 10 sec or higher. Although in reality, it is possible to schedule the job on a resource with an available time interval equal to 5 sec or more.

The rest of this chapter is organized as follows. The focus of Section 6.1 is on describing a *Prescheduling Error Handling* (PSEH) technique, which attempts to correct the error in user-estimated execution times before the job is scheduled. In Section 6.2, a description of the prototyping and measurement experiments conducted to evaluate the effectiveness of the PSEH technique is provided. The results of the performance evaluation and insights gained into system behaviour are discussed in Section 6.3. Lastly, Section 6.4 provides a summary and discussion of the chapter.

6.1 Prescheduling Error Handling Technique

The PSEH technique uses the past history of the jobs submitted to the system to establish a trend for the error in the user's estimated execution times. The objective of the PSEH technique is to correct the error in the user-estimated execution times to make them more accurate before the job is passed on to the resource management algorithm for scheduling. More specifically, when there are overestimated execution times, the PSEH technique decreases the user-estimated execution times, and when there are underestimated execution times, the PSEH technique increases the user-estimated execution times. The resource management algorithm can then use the adjusted execution times to perform matchmaking and scheduling. A discussion of the PSEH technique and how it is incorporated into the HCP-RM algorithm is presented next. Note that the PSEH technique can also be adapted and used by other resource management algorithms.

The PSEH technique uses two variables: `avgMapTaskErrorFactor` and `avgReduceTaskErrorFactor` to keep track of how much the user-estimated map task execution times and the user-estimated reduce task execution times, respectively, are being overestimated or underestimated. Note that the term *average task error factor* is used to refer to both `avgMapTaskErrorFactor` and `avgReduceTaskErrorFactor` collectively. The average task error factor is initialized to 0, meaning that there is currently no error in the execution times in the system. If the average task error factor is greater than 0, it means that on average the user-estimated execution times are being overestimated. On the other hand, if the average task error factor is less than 0, it means that on average the user-estimated task execution times are being underestimated. The closer the average task error

factor is to 0, the more accurate the user-estimated task execution times are to the actual runtimes of the tasks.

The average task error factor is calculated in a new method named `calculateTaskErrorFactor()`. This method is invoked each time a task completes its execution and its purpose is to calculate the *error factor of the task* and then recalculate the average task error factor. The error factor of a task t (denoted f_t) is calculated as follows:

$$f_t = \frac{e_t^{est} - e_t^{run}}{e_t^{run}} \tag{6.1}$$

where e_t^{est} is the user-estimated execution time of task t and e_t^{run} is the actual runtime of task t. The value of f_t determines how accurate the original user-estimated execution time of task t is to the actual runtime of task t. If the task is a map task, the `avgMapTaskErrorFactor` is updated by including the f_t of the recently completed task t in the calculation of the average. More specifically, the `avgMapTaskErrorFactor` is equal to the sum of the f_t of all the completed map tasks divided by the total number of completed map tasks. On the other hand, if the task is a reduce task, the `avgReduceTaskErrorFactor` is updated in a similar manner as described for the `avgMapTaskErrorFactor`. It is expected that overtime as more jobs complete executing, the average task error factor will get more accurate, resulting in the adjusted task execution times being closer to the actual runtimes of the tasks.

The PSEH technique also uses a method called `adjustExecutionTime()` to adjust the user-estimated task execution time and make the execution times more accurate. Each map task and reduce task has its execution time adjusted before it is scheduled on the system. The following equation is used to calculate the *adjusted task execution time* of a task t (e_t^{adj}):

$$e_t^{adj} = \frac{e_t^{est}}{1 + f} \tag{6.2}$$

where e_t^{est} is the user-estimated task execution time and f is the execution time error factor. Depending on whether the task t is a map task or a reduce task, f can either be equal to the `avgMapTaskErrorFactor` or the `avgReduceTaskErrorFactor`.

6.2 Performance Evaluation of the PSEH Technique

A performance evaluation of the PSEH technique is conducted using prototyping and measurement to determine its effectiveness in improving system performance when there is error in the user-estimated execution times. The experiments are performed on a Hadoop cluster deployed on Amazon EC2. More specifically, the performance of HCP-RM that uses the PSEH technique (denoted HCP-RM-EH) is compared with the performance of the original version of HCP-RM.

The rest of this section is organized as follows. In Section 6.2.1, the experimental setup, including the metrics used in the performance evaluation, is described. Next, in Section 6.2.2, a description of the workload that is used in the experiments is provided. Lastly, Section 6.2.3 describes the models used to generate the error/inaccuracies in the user-estimated execution times.

6.2.1 Experimental Setup

The Hadoop cluster that is used to conduct the experiments is the same as the one described in Section 5.5.1. In addition, the following metrics are used in the experiments:

- *Proportion of late jobs* (*P*) (recall Section 4.4.1)
- *Average job turnaround time* (*T*) (recall Section 4.4.1)

- *Average job matchmaking and scheduling time* (*O*): *O* is measured using Java's `System.nanoTime()` [102] method and is calculated as the total processing time required by the respective resource management algorithm (e.g., HCP-RM or HCP-RM-EH) to match make and schedule jobs in an experiment, divided by the total number of jobs arriving on the system during the experiment.

6.2.2 System and Workload Parameters

The workload that is used in the experiments is the Hadoop Synthetic Workload described in Section 5.5.3 using the default values for the workload parameters. This is the same workload that is used in the experiments performed to investigate the effect of error in user-estimated execution times on the performance of the HCP-RM technique (see Section 5.7). The experiments are conducted using a synthetic workload because they allow the execution times to be methodically and accurately controlled, allowing the effectiveness of the PSEH technique to be evaluated systematically.

6.2.3 Models for Generating Error in User-estimated Execution Times

In order to generate the error in user-estimated task execution times, three models are used: (1) Constant Error Model, (2) Feitelson's Error Model, and (3) Variable Error Model. Recall that the Constant Error Model and Feitelson's Error Model were already described in Section 5.7.1. A description of the Variable Error Model is provided next.

The *Variable Error Model* is an extension of the Constant Error Model, and it is devised to investigate the effectiveness of the PSEH technique when not all the jobs submitted to the system have the same execution time error factor (*f*). As described in Section 5.7.1, when using the Constant Error Model, all the jobs submitted to the system have the same value of *f*; however, when using the Variable Error Model each job submitted

to the system can have a different value of *f*. Algorithm 6.1 describes how the Variable Error Model generates *f* for each job submitted to the system. The input parameters required by the algorithm are described next.

- *poe*: the probability that a job has an overestimated execution time (i.e., *f* is greater than 0). The value of *poe* must be in the interval [0, 1].
- *min_ue_f* and *max_ue_f*: the minimum value and the maximum value of the uniform distribution used to generate the error factor for jobs with *underestimated* execution times. The values of *min_ue_f* and *max_ue_f* must be in the interval (-1, 0], and *max_ue_f* must also be greater than or equal to *min_ue_f*.
- *min_oe_f* and *max_oe_f*: the minimum value and the maximum value of the uniform distribution used to generate the error factor for jobs with *overestimated* execution times. The values of *min_oe_f* and *max_oe_f* must be greater than or equal to 0, and *max_oe_f* must also be greater than or equal to *min_oe_f*.

Algorithm 6.1: Variable Error Model

Input: *poe*, *min_ue_f*, *max_ue_f*, *min_oe_f*, *max_oe_f*
Output: execution time error factor, *f*

```
1:  rv ← Generate a uniformly distributed random variable from [0, 1].
2:  if rv <= poe then
3:      f ← Generate a uniformly distributed random variable from
            [min_oe_f, max_oe_f].
4:  else
5:      f ← Generate a uniformly distributed random variable from
            [min_ue_f, max_ue_f].
6:  end if
7:  return f
```

The first step of Algorithm 6.1 is to generate a uniformly distributed random value from the interval [0, 1] and store it in a variable named *rv* (line 1). If the value of *rv* is less than or equal to the value of *poe*, a positive execution time error factor is generated using

a uniform distribution within the interval [*min_oe_f*, *max_oe_f*] (lines 2-3). Otherwise, *rv* is greater than *poe* and a negative execution time error factor is generated using a uniform distribution within the interval [*min_ue_f*, *max_ue_f*] (lines 4-5). The last step is to return the error factor that is generated (line 7). Now that an error factor for the job has been generated, the estimated execution times of the job's tasks can be calculated in a similar manner as the Constant Error Model (see Eq. 5.1 in Section 5.7.1).

Relationship between User-estimated Job Execution Times and the Deadlines of Jobs: As shown in Table 5.1, the deadline of a job *j* in the Hadoop Synthetic Workload is calculated as follows: $d_j = s_j + SET_j^R * em$ where s_j is the earliest start time of job *j*, SET_j^R is the estimated execution time of job *j* when it executes at its maximum degree of parallelism on a set of resources *R*, and *em* is the execution time multiplier, which is used to determine the laxity of the job. As described in Section 5.7.2, the deadline of each job submitted to the system is calculated using the respective job's estimated task execution times, which may contain error. The rationale behind this decision is that a user who submits a job will generate a deadline for his/her job using the estimated task execution times that he/she supplies to the system. Thus, if the job has *overestimated* task execution times, the job will have *more* laxity (or slack time) compared to when there is no error in the execution times. On the other hand, if the job has *underestimated* task execution times, the job will have *less* laxity (or slack time) compared to when the execution times have no error.

6.3 Results of the Performance Evaluation

This section presents and discusses the results of the experiments performed to evaluate the PSEH Technique. To generate each of the values shown in the graphs and

tables presented in this section, the experiments are run long enough to ensure the system reached a steady state, where each experiment lasted approximately 24 hours (similar to the performance evaluation described in Section 5.5.1). Note that in this section, Figure 6.5 and Figure 6.7 display the values of T and O in the same figure, where T is displayed as a bar graph that uses the scale on the left Y-axis and O is displayed as a sequence of points that uses the scale on the right Y-axis.

6.3.1 Constant Error Model

Figure 6.1 and Table 6.1 present a comparison of P between the HCP-RM technique that uses the PSEH technique (HCP-RM-EH) and the original version of the HCP-RM technique when the Constant Error Model is used with different values of f and λ. Note that the results for HCP-RM are the same as those presented in Section 5.7.3 and are replicated here to compare with the results of HCP-RM-EH. For a given λ, it is observed that both HCP-RM-EH and HCP-RM follow a similar trend in performance: P decreases as f increases due to jobs having higher values of laxity captured in L_j^{act} (defined in Section 5.7.2), resulting in less stringent deadlines. For all the values of λ and values of f experimented with, it is observed that HCP-RM-EH achieves a lower or equal P compared to HCP-RM. More specifically, HCP-RM-EH achieves up to an approximately 50% reduction in P (when λ is 1/25 jobs per sec and f is -0.5 as shown in Table 6.1) and on average a 29% reduction in P over all the values of f and λ experimented with. Furthermore, it is observed that in some cases, such as when λ is 1/30 jobs per sec and f is 2, HCP-RM-EH achieves a P of 0 (as indicated by the missing bar corresponding to the technique in Figure 6.1).

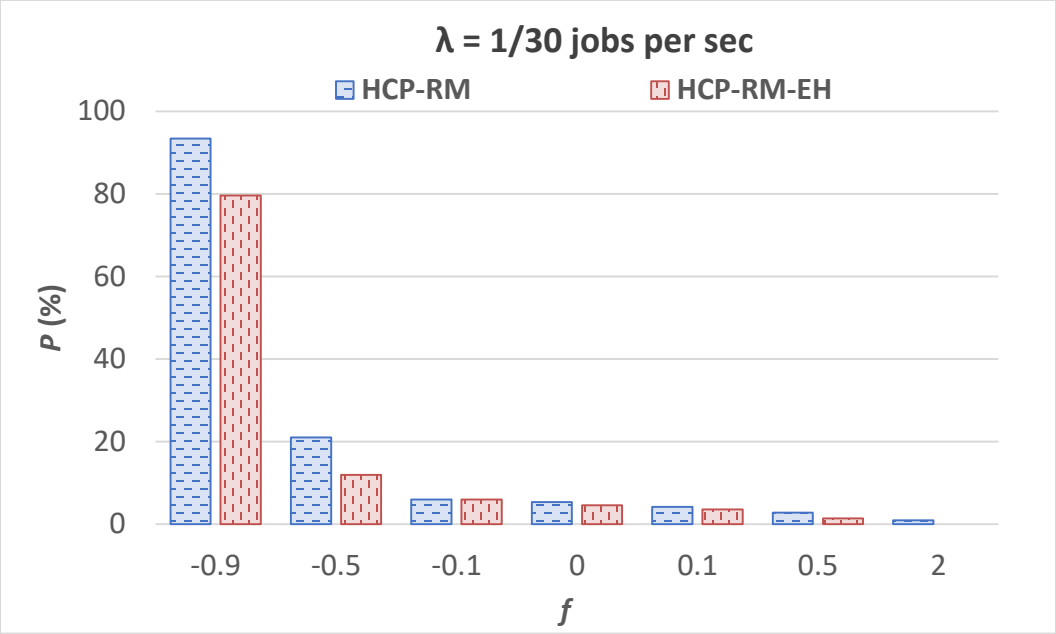

Figure 6.1. HCP-RM vs HCP-RM-EH: effect of *f* on *P* when using the Constant Error Model and λ is 1/30 jobs per sec.

The superior performance of HCP-RM-EH can be attributed to the ability of the PSEH technique to adjust the user-estimated task execution times and make them more accurate. This in turn allows HCP-RM-EH to make intelligent matchmaking and scheduling decisions that lead to a small *P*. Having accurate execution times is even more important at higher values of λ due to the high contention for resources, and thus, it is observed that the overall performance improvement (in terms of *P*) of HCP-RM-EH over HCP-RM increases as λ increases (see Table 6.1). Moreover, it is observed that for a given λ, the performance improvement of HCP-RM-EH over HCP-RM is at its highest when *f* is less than or equal to -0.5 (see Figure 6.1, for example). This is because at these low (negative) values of *f*, the job execution times are significantly underestimated when the schedule is prepared by HCP-RM. During runtime, the actual execution times are significantly longer, resulting in job completion times that are higher compared to the

values computed for the schedule. This leads to a higher chance of a deadline miss for HCP-RM. The PSEH technique used by HCP-RM-EH attempts to correct the error in job execution times, resulting in more accurate execution times used by HCP-RM-EH when the schedule is prepared, thus leading to a lower chance of deadline misses.

Table 6.1. HCP-RM vs HCP-RM-EH: effect of *f* on *P* when using the Constant Error Model.

f	**λ = 1/45 jobs per sec**		**λ = 1/25 jobs per sec**	
	HCP-RM: *P* (%)	**HCP-RM-EH: *P* (%)**	**HCP-RM: *P* (%)**	**HCP-RM-EH: *P* (%)**
-0.9	87.2	64.4	95.2	91.8
-0.5	12.6	8	66.8	34
-0.1	2.8	2.6	26.6	14.4
0	2.8	2.6	19.2	11.4
0.1	2	1.6	18	12.8
0.5	1	1	6.8	6.8
2	0.2	0	1.8	1.6

A comparison of the values of *T* achieved by HCP-RM-EH and HCP-RM using the Constant Error Model at different values of λ and *f* are shown in Figure 6.2 and Table 6.2. The results show that HCP-RM-EH tends to achieve a lower *T* compared to HCP-RM. The highest (57%) reduction in *T* is observed when λ is 1/25 jobs per sec and *f* is -0.9. Over all the experiments conducted, HCP-RM achieves on average a 21% reduction in *T*. Once again, the improved performance of HCP-RM-EH can be attributed to the PSEH technique generating more accurate estimated task execution times, which enables more effective matchmaking and scheduling decisions to be made.

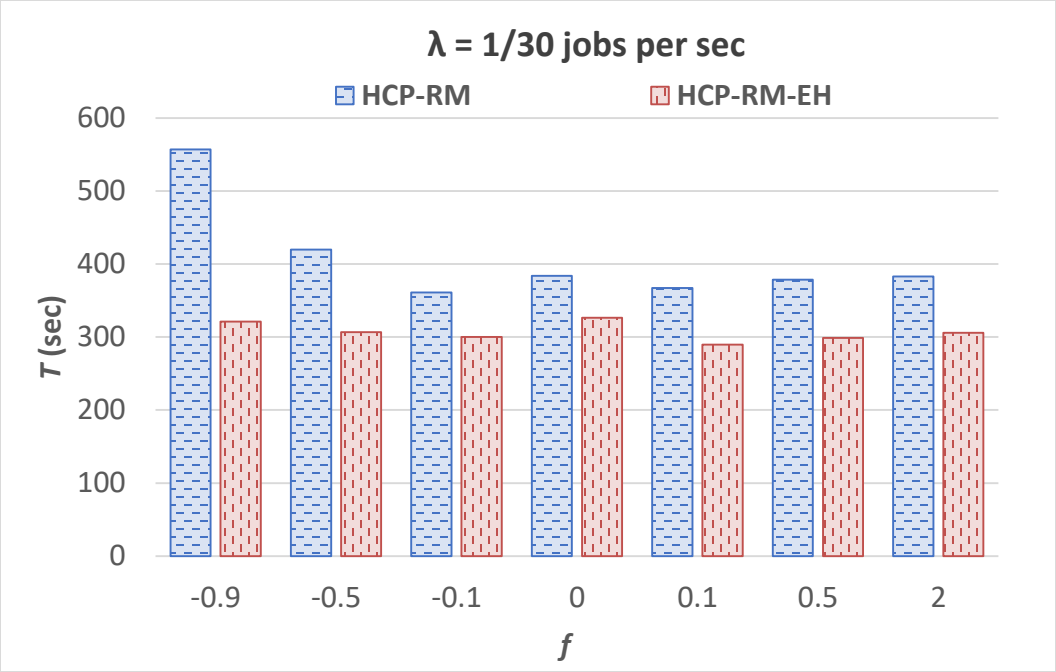

Figure 6.2. HCP-RM vs HCP-RM-EH: effect of *f* on *T* when using the Constant Error Model and λ is 1/30 jobs per sec.

Moreover, it is also observed that for a given λ, the trend for *T* for the two systems are quite different. In the case of HCP-RM, *T* tends to increase as *f* decreases, whereas in the case of HCP-RM-EH, *T* tends to remain at approximately the same value as *f* changes. In the case of HCP-RM-EH, it is expected that *T* would not significantly change with *f* because for each experiment where the value of *f* is varied, the actual runtimes of the jobs in the workload that need to be processed is the same, only the user-estimated execution times of the jobs are different. Thus, for all the values of *f* and a given value of λ, HCP-RM-EH is processing a similar workload in each experiment. On the other hand, in the HCP-RM experiments, the user-estimated execution times are used directly to perform matchmaking and scheduling, leading to HCP-RM seeing a different workload that needs to be processed in each experiment where *f* is changed. Note that the explanation for the trend in *T* for HCP-RM is described in more detail in Section 5.7.3.

Table 6.2. HCP-RM vs HCP-RM-EH: effect of *f* on *T* when using the Constant Error Model.

f	λ = 1/45 jobs per sec		λ = 1/25 jobs per sec	
	HCP-RM: *T* (sec)	**HCP-RM-EH: *T* (sec)**	**HCP-RM: *T* (sec)**	**HCP-RM-EH: *T* (sec)**
-0.9	294	202	1513	650
-0.5	246	199	1249	664
-0.1	221	198	956	670
0	216	199	884	679
0.1	218	200	834	681
0.5	224	197	724	660
2	229	196	608	717

Figure 6.3 and Table 6.3 present a comparison of *O* between HCP-RM-EH and HCP-RM when using the Constant Error Model at different values of *f* and λ. Similar to the results of *T*, it is observed that the trends in *O* achieved by HCP-RM-EH and HCP-RM are different. For HCP-RM, it is observed that *O* tends to increase with *f*. Recall from Section 5.7.3 that this is due to the user-estimated job execution times being directly proportional to *f*. As the user-estimated job execution times increase, HCP-RM expects a higher contention for resources that require the solver to spend more time in finding a solution to the resource management problem. This leads to an increase in *O*. On the other hand, for HCP-RM-EH, it is observed that *O* tends to remain at approximately the same value as *f* is varied. Similar to the explanation provided for the results of *T*, this trend in performance can be attributed to the fact that in each experiment where *f* is varied, the actual runtimes of the jobs in the workload that need to be processed is the same, only the user-estimated execution times of the jobs are different. In addition, the PSEH technique is able to adjust the user-estimated execution times and make them closer to the actual runtimes. As a result, the overhead of the CPLEX solver is relatively insensitive to the value of *f*. The highest *O* is observed when *f* is -0.9. This is because jobs have less laxity

and thus are more susceptible to miss their deadlines. This in turn causes HCP-RM-EH to require more time to find a schedule that minimizes *P*.

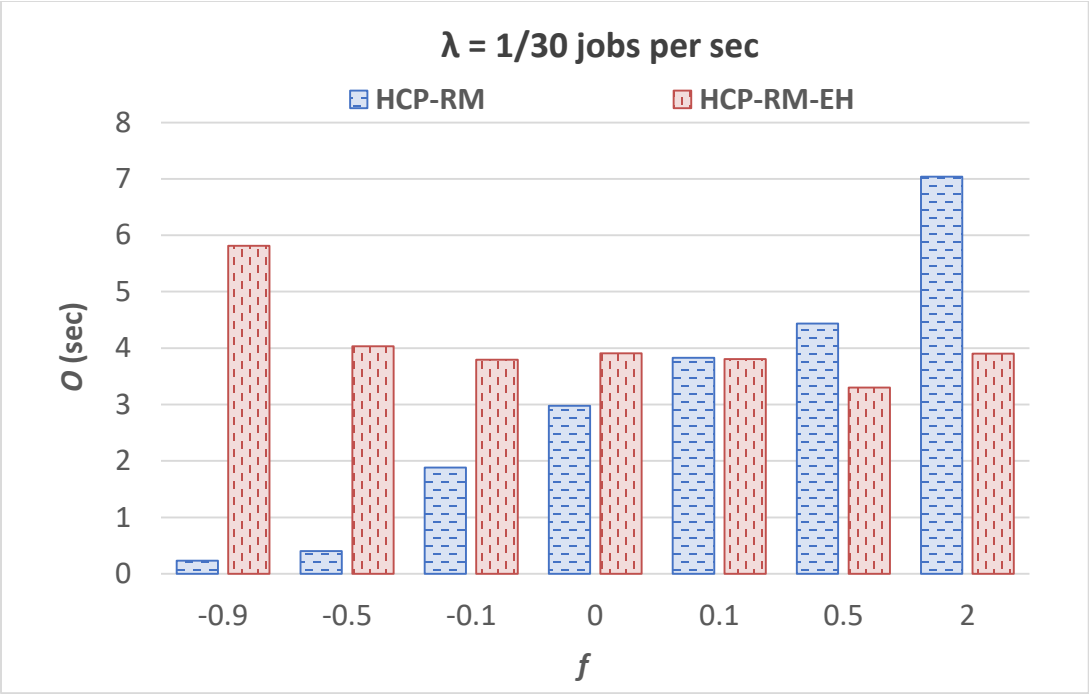

Figure 6.3. HCP-RM vs HCP-RM-EH: effect of *f* on *O* when using the Constant Error Model and λ is 1/30 jobs per sec.

Table 6.3. HCP-RM vs HCP-RM-EH: effect of *f* on *O* when using the Constant Error Model.

f	**λ = 1/45 jobs per sec**		**λ = 1/25 jobs per sec**	
	HCP-RM: *O* (sec)	**HCP-RM-EH: *O* (sec)**	**HCP-RM: *O* (sec)**	**HCP-RM-EH: *O* (sec)**
-0.9	0.17	2.20	1.34	12.80
-0.5	0.24	1.27	3.37	11.36
-0.1	0.67	1.58	5.75	11.95
0	0.74	1.35	8.09	10.79
0.1	0.87	1.43	9.31	10.72
0.5	1.65	1.07	12.60	10.42
2	2.19	1.12	15.19	11.50

Another interesting observation from the results shown in Table 6.3 and Figure 6.3 is that when *f* is less than 0 (i.e., execution times are underestimated), HCP-RM achieves a lower *O* compared to HCP-RM-EH. When *f* is less than 0, the job execution times used by HCP-RM are lower than their actual execution times. The PSEH technique used by HCP-RM-EH tends to correct the underestimated job execution times, resulting in the respective jobs processed by HCP-RM-EH to have higher execution times compared to those handled by HCP-RM. As a result, since higher job execution times lead to a higher resource contention, the CPLEX solver in HCP-RM-EH takes a longer time to generate a schedule for the system in comparison to the solver in HCP-RM. On the other hand, when task execution times are significantly overestimated (e.g., *f* is greater than or equal to 0.5), HCP-RM tends to have a higher *O* compared to HCP-RM-EH. This is because the solver in HCP-RM needs to handle a higher resource contention in comparison to HCP-RM-EH that processes jobs with lower execution times due to the PSEH technique tending to correct the overestimated execution times, resulting in a lower contention for resources. This leads to a lower processing time for the CPLEX solver in the case of HCP-RM-EH, and thus HCP-RM-EH achieves a lower *O* compared to HCP-RM.

6.3.2 Feitelson's Error Model

The results of the performance comparison between HCP-RM-EH and HCP-RM when using Feitelson's Error Model (described in Section 5.7.1) at different arrival rates are presented in Figure 6.4 and Figure 6.5. As shown in Figure 6.4, it is observed that both HCP-RM-EH and HCP-RM achieve the same values of *P* for all the values of λ experimented with. This can be attributed to Feitelson's Error Model generating highly overestimated task execution times, resulting in jobs having more slack time (i.e., higher

values of L_j^{act}) and lenient job deadlines. Note that the non-visible bars in Figure 6.4 when λ is 1/45 jobs/sec and 1/30 jobs per sec indicate that the value of P is 0 for both systems.

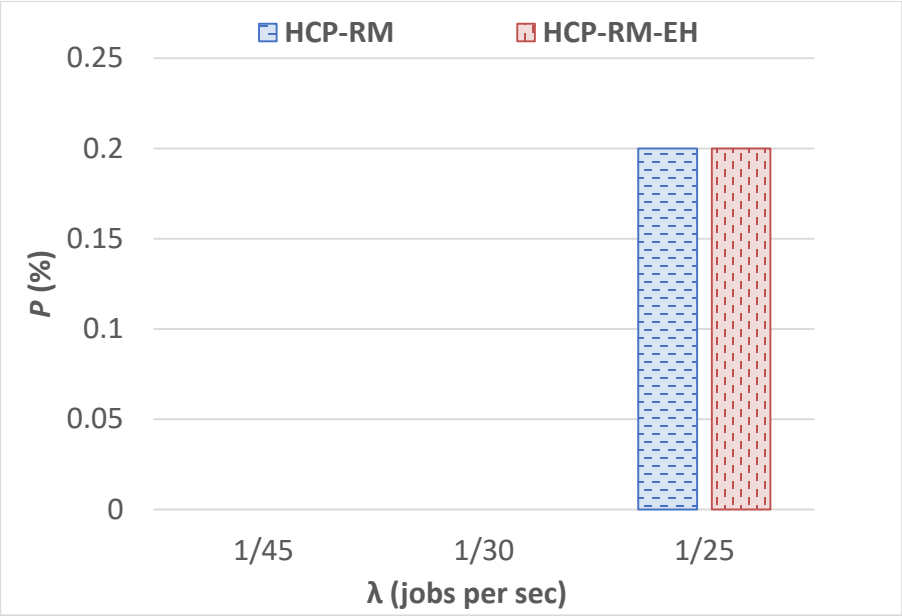

Figure 6.4. HCP-RM vs HCP-RM-EH: effect of λ on P when using Feitelson's Error Model.

Figure 6.5. HCP-RM vs HCP-RM-EH: effect of λ on T and O when using Feitelson's Error Model.

With regards to *T* (refer to Figure 6.5), it is observed that for a given λ, both HCP-RM-EH and HCP-RM have comparable values with HCP-RM achieving a slightly lower *T*. This can be attributed to HCP-RM expecting that jobs have very high execution times, and thus focusing on executing jobs at or close to their arrival times so that the jobs do not miss their deadlines. On the other hand, HCP-RM-EH sees jobs that have lower execution times (as adjusted by the PSEH technique) and more laxity in comparison to HCP-RM. This allows HCP-RM-EH to quickly find a schedule that minimizes *P* without focusing on minimizing *T*, leading HCP-RM-EH to have a slightly higher *T* compared to HCP-RM. This reasoning is supported by observing the results of *O* shown in Figure 6.5 (refer to the sequence of points), which demonstrate that HCP-RM-EH achieves a lower *O* (on average 72% lower) compared to HCP-RM for all the values of λ experimented with. HCP-RM expects a high contention for resources because when using Feitelson's Error Model, it receives jobs with highly overestimated execution times. This in turn leads to more processing time being required to match make and schedule jobs and results in a higher *O*.

6.3.3 Variable Error Model

Figure 6.6 and Figure 6.7 present the results of the performance comparison between HCP-RM-EH and HCP-RM when using the Variable Error Model. The parameters of the Variable Error Model (defined in Section 6.2.3) are set as follows: *poe* = 0.98, *min_ue_f* = -0.9, *max_ue_f* = -0.1, *min_oe_f* = 0, *max_oe_f* = 2. The value of *poe* is adopted from [77], which performed an analysis on a workload trace and found that 98% of jobs submitted have overestimated execution times and only 2% have underestimated execution times. The values of *min_ue_f*, *max_ue_f*, *min_oe_f*, and *max_oe_f* are set

according to the maximum and minimum values of f used in the Constant Error Model experiments (discussed in Section 6.3.1).

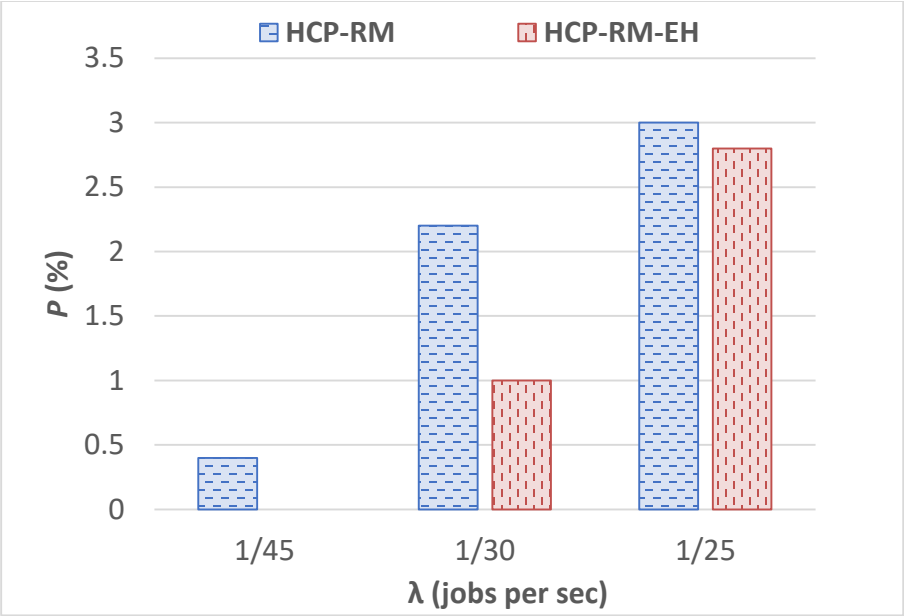

Figure 6.6. HCP-RM vs HCP-RM-EH: effect of λ on P when using the Variable Error Model.

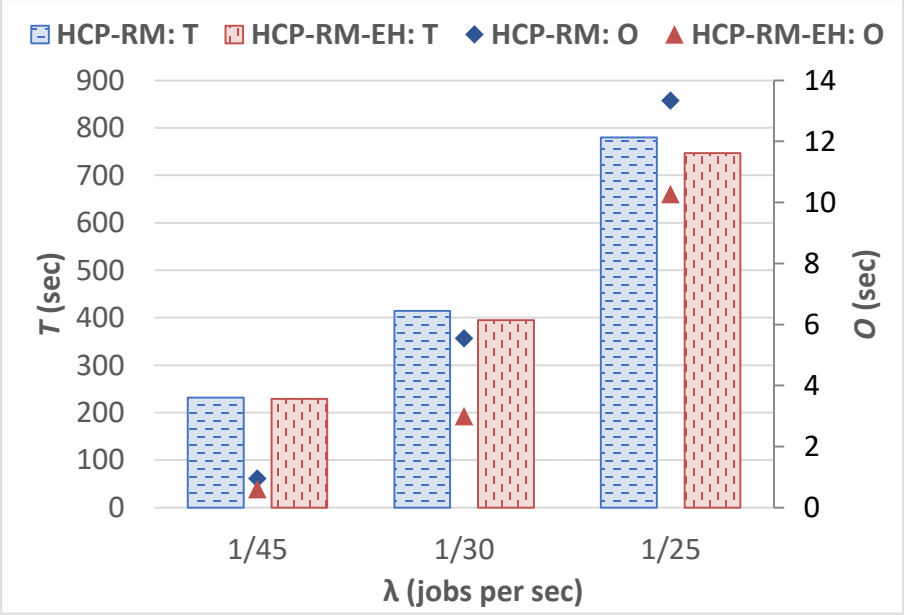

Figure 6.7. HCP-RM vs HCP-RM-EH: effect of λ on T and O when using the Variable Error Model.

It is observed that HCP-RM-EH achieves a lower *P*, *T*, and *O* compared to HCP-RM for all the values of λ experimented with. Note that the non-visible bar for HCP-RM-EH in Figure 6.6 when λ is 1/45 jobs/sec indicates that *P* is 0. The performance improvement in terms of *P*, *T*, and *O* achieved by HCP-RM-EH over HCP-RM when using the Variable Error Model is summarized: 54% reduction in *P*, 3% reduction in *T*, and 35% reduction in *O*. The results demonstrate that the PSEH technique used by HCP-RM-EH is effective in not only handling workloads in which jobs have the same *f* as generated by the Constant Error Model, but is also effective in handling workloads in which jobs have different values of *f* as generated by the Variable Error Model. The superior performance of HCP-RM-EH can be attributed to the PSEH technique being able to adjust the user-estimated task execution times and make them more accurate (i.e., closer to the actual task runtimes). This gives HCP-RM-EH more accurate information on how long jobs need to execute for, and thus, allows HCP-RM-EH to make more intelligent matchmaking and scheduling decisions that can lead to high system performance.

6.4 Summary and Discussion

In this chapter, techniques are presented for handling inaccuracy or error in user estimates of job execution times (submitted as part of the SLA for the job) to improve the robustness of the MRCP-RM technique (described in Chapter 4) and the HCP-RM technique (described in Chapter 5). The effectiveness of a matchmaking and scheduling algorithm that depends on the user-estimated job execution times can be diminished by inaccurate estimates of job runtimes. Thus, a Prescheduling Error Handling (PSEH) technique is devised to adjust the user-estimated execution times to make them more accurate before they are used by the resource management algorithms. A rigorous

performance evaluation of the PSEH technique is conducted on a Hadoop cluster deployed on Amazon EC2. Three models to generate the error in user-estimated execution times (Constant, Feitelson's, and Variable) are used in the experiments. The performance of HCP-RM using the PSEH technique (denoted HCP-RM-EH) is compared to the performance of the original version of HCP-RM. A number of insights into system behaviour and performance are gained from analyzing the results of the experiments and these are summarized next.

- *Constant Error Model*: In all the experiments conducted using the Constant Error Model (see Section 6.3.1), HCP-RM-EH is observed to achieve a lower P (up to 50% lower and on average 29% lower) and a lower T (up to 57% lower and on average 21% lower) compared to HCP-RM. Furthermore, HCP-RM-EH achieves a P of 0 when f is 2 and λ is 1/30 jobs per sec or lower. With regards to O, it is observed that when the execution times are overestimated, HCP-RM-EH achieves a lower O compared to HCP-RM, but when the execution times are underestimated, the opposite is true.
- *Feitelson's Error Model*: When using Feitelson's Error Model (see Section 6.3.2), both HCP-RM-EH and HCP-RM achieve the same values of P of less than 0.2% and similar values of T due to Feitelson's Error Model generating jobs with significantly overestimated execution times. However, HCP-RM-EH is observed to achieve up to a 76% lower O (on average 72% lower) compared to the O achieved by HCP-RM.
- *Variable Error Model*: Over all the experiments conducted using the Variable Error Model (see Section 6.3.3), it is observed that HCP-RM-EH achieves on

average a 54% lower *P*, a 3% lower *T*, and a 35% lower *O* compared to the values of *P*, *T*, and *O*, respectively, achieved by HCP-RM. This demonstrates the effectiveness of HCP-RM-EH in handling a workload comprising jobs with different degrees of errors in execution times.

- *Effectiveness of PSEH technique*: The superior performance of HCP-RM-EH can be attributed to the PSEH technique being able to adjust the user-estimated task execution times to make them more accurate. This in turn enables HCP-RM-EH to make intelligent matchmaking and scheduling decisions that tends to lead to HCP-RM-EH achieving lower values of *P*, *T*, and *O* compared to HCP-RM.

In the following sub-section, a direction for future research that focuses on devising a *runtime error handling technique*, which deals with the error in execution times after the job has been scheduled and has started running, is described.

6.4.1 Runtime Error Handling Technique

The PSEH technique described in this chapter alters the job execution times (before the job is scheduled on the system) based on the trend of error in user-estimated execution times, which is established from the past history of completed jobs. The resulting adjusted values of such execution times may still not be 100% accurate. Thus, there seems to be a scope for future research on a technique that performs further error handling after the jobs start running on the system. Such a technique will be targeted at handling two situations: (1) when the system generated execution times determined by the PSEH technique are still overestimated, leading to resource idle times, and (2) when the system generated execution times are still underestimated, preventing jobs from being able to complete executing by their computed end times. Handling of each of these situations is briefly discussed.

Handling of Overestimated Execution Times: This corresponds to the situation in which task *t* takes less than its estimated execution time (e_t^{est}) to complete. In this situation, the completed task *t* is removed from the system and the tasks scheduled on the resource *r*, on which *t* was running, are rescheduled so that they can start to execute earlier as long as their earliest start time requirements are not violated. This attempts to decrease the resource idle time that will result from task *t* completing before its expected end time.

Handling of Underestimated Execution Times: This corresponds to the situation in which task *t* does not complete its execution after running for e_t^{est} time units. In such a situation, task *t* can be given additional execution time (referred to as a *time quantum*) so that it can continue executing immediately or at a later time. The system can start by assigning task *t* a small time quantum as long as it does not lead to new deadline misses for jobs. If the task still does not finish executing, further time quanta with higher durations can be provided if the deadline for the job containing task *t* is not violated. The idea is to utilize existing idle slots in the resource schedule and to delay executing the tasks of jobs that have already missed their deadlines in favour of executing tasks of jobs that have not missed their deadlines. Whether using such an additional runtime error handling technique leads to a significant improvement in system performance is worthy of further investigation.

Chapter 7 Workflow Budget-Based Resource Management Technique

The focus of this chapter is on describing the Workflow Budget-Based Resource Management (WFBB-RM) technique that is devised to efficiently 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. Preliminary research on a budget-based resource allocation and scheduling technique for processing an open stream of MapReduce jobs with SLAs is described in [19]. In addition to MapReduce jobs, the WFBB-RM technique can process workflows that have different structures (various types of precedence relationships) and more than two phases of execution, such as scientific workflows found in the domain of physics and biology. The WFBB-RM technique decomposes (*budgets*) the end-to-end deadline of a job, which is submitted as part of the job's SLA, into components (i.e., sub-deadlines), each of which is associated with a specific 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 minimizing the number of jobs that miss their deadlines.

The rest of the chapter is organized as follows. Section 7.1 provides a description of how the resource allocation and scheduling problem is modelled. The algorithms devised to budget the end-to-end deadline for multi-stage jobs are described in Section 7.2. In Section 7.3, the matchmaking and scheduling algorithms of the WFBB-RM technique are discussed. The experimental setup and description of the workloads used in the performance evaluation of the WFBB-RM technique are described in Section 7.4. The results of the experiments are presented and the insights gained into system behaviour and

performance are discussed in Section 7.5 and Section 7.6. Lastly, Section 7.7 provides a summary and discussion of the chapter.

7.1 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 Figure 7.1). 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, $R = \{res\ 1,\ res\ 2, \ldots, res\ m\}$ 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 related works have modelled resources in a similar manner (see [36], [39], and [41], for example).

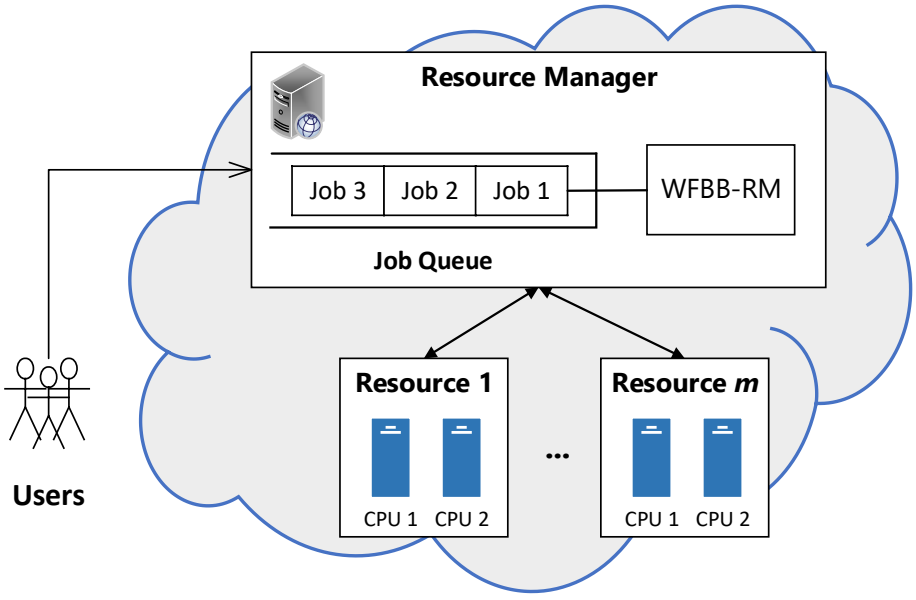

Figure 7.1. Example of a system deploying the WFBB-RM technique.

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 Figure 7.2, 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.

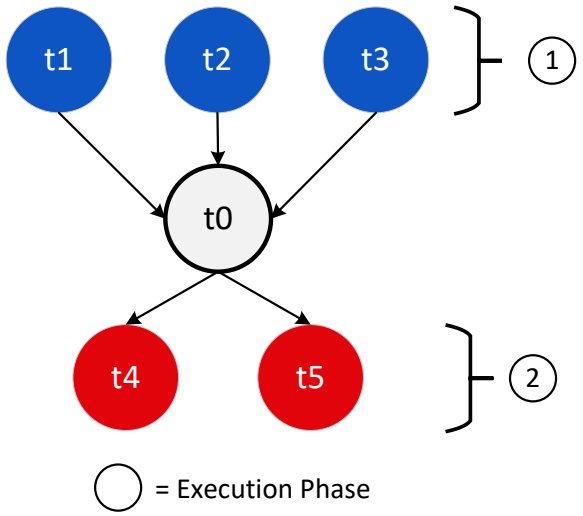

Figure 7.2. DAG of a sample multi-stage job.

The example multi-stage job shown in Figure 7.2 is characterized by two phases of execution. An *execution phase* in a multi-stage job comprises one or more tasks that can only start to execute after the previous execution phase is completed, which also comprises a single task or set of tasks. The execution phases of a multi-stage job can only be executed sequentially. For example, in the sample job shown in Figure 7.2, the first phase of execution comprises three tasks: t1, t2, and t3. These three tasks do not have any *direct*

preceding tasks (referred to as *parent tasks*), which means that these tasks can start executing at the job's earliest start time. However, the tasks t4 and t5, which are part of the second phase of execution, each have a parent task t0, as well as *indirect preceding tasks* t1, t2, and t3. The tasks t4 and t5 cannot 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 show precedence relationships between tasks in the DAG, and thus, dummy tasks have an execution time equal to 0. For example, in Figure 7.2, 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 Figure 7.1, 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 the WFBB-RM technique 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 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 the WFBB-RM algorithm. However, a task that has been scheduled but has not started executing can be rescheduled or assigned to another resource, if required.

7.2 Deadline Budgeting Algorithm for Workflows

Algorithm 7.1 presents the *Deadline Budgeting Algorithm for Workflows* (abbreviated DBW), which is used by the WFBB-RM technique 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 parameters: `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. Recall the discussion of SET_j and L_j provided in Section 3.1.1. SET_j is an estimate of the execution time of job j and L_j is the extra time that job j has for meeting its deadline if it starts executing at its earliest start time: $L_j = d_j - s_j - SET_j$.

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` parameter. 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 (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 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

$SET_j^{R\_PL}$). Next, the algorithm calculates the laxity of the job (L_j) as shown in 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. 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 for 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 Algorithm* (abbreviated PD), which is described in Section 7.2.2, and (2) the *Even Distribution of Job Laxity Algorithm* (abbreviated ED), which is discussed in Section 7.2.3. The algorithm that is used depends on the supplied `laxDistOpt` input parameter. Before discussing the details of the PD and ED algorithms, a discussion on the laxity of tasks is presented first.

Algorithm 7.1: 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:** $jobLaxity \leftarrow d_j - s_j - SET_j$ **3:** According to *laxDistOpt*, invoke the PD algorithm or the ED algorithm.

7.2.1 Laxity of Tasks

After invoking the DBW algorithm for a job *j*, each task *t* belonging to job *j* will have a sub-deadline. Therefore, the laxity of a task *t*, denoted LT_t, can be calculated as follows:

$$LT_t = sd_t - ts_t - e_t \tag{7.1}$$

where sd_t is the sub-deadline of task t, ts_t is the earliest start time of task t, and e_t is the execution time of task t. The value of ts_t is dependent on the precedence relationships of task t. For tasks that do not have any parent tasks (recall the definitions provided in Section 7.1), ts_t is equal to s_j (the earliest start time of the job that t belongs to). If a task t has at least one parent task, ts_t is equal to the completion time of the latest finishing parent task of t. The value of LT_t is between a *minimum task laxity* value (LT_t^{min}) and a *maximum task laxity* value (LT_t^{max}).

LT_t^{max} is the maximum laxity that task t can have and is calculated as follows:

$$LT_t^{max} = sd_t - eps_t - e_t \tag{7.2}$$

where eps_t is the earliest possible start time of task t when the parent job of t is executed on the set of resources R (comprising m resources). The value of eps_t is equal to the completion time of the latest finishing parent task of t, given that t's parent tasks finish executing at their earliest possible times (i.e., none of t's preceding tasks (direct or indirect) use any of their laxities). Moreover, LT_t^{min} is the minimum laxity that task t can have. In other words, LT_t^{min} is the laxity of task t given that all of t's parent tasks complete their execution at their respective sub-deadlines (i.e., every parent task of t uses all of its laxity). LT_t^{min} is calculated as follows:

$$LT_t^{min} = sd_t - sd_t^{LFPT} - e_t \tag{7.3}$$

where sd_t^{LFPT} is equal to the sub-deadline of the latest finishing parent task of t. Note that if a task has no parent tasks, sd_t^{LFPT} is equal to s_j.

7.2.2 Proportional Distribution of Job Laxity Algorithm

The Proportional Distribution of Job Laxity Algorithm (abbreviated PD) 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 PD algorithm is shown in Algorithm 7.2. The input required by the algorithm includes a job j to process and an integer parameter, `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.

Algorithm 7.2: 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 setParentTasksSubDeadlines(t)
10:     end if
11: end for
```

The first step of Algorithm 7.2 is to calculate the sample completion time of job j (denoted SCT_j) as: $s_j + SET_j$ 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 follows:

$$CL_t = \frac{SCT_t - s_j}{SCT_j - s_j} \times L_j \quad (7.4)$$

where SCT_t is the sample completion time of task t. Note that the sample completion time of the tasks are determined during the calculation of SET_j (line 1 in Algorithm 7.1). The cumulative laxity of a task t is the maximum laxity that task t can have (recall Section 7.2.1). After calculating CL_t, the sub-deadline of the task t (sd_t) is then calculated (line 6) as follows:

$$sd_t = SCT_t + CL_t \quad (7.5)$$

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). This method, whose objective 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, is described in more detail in Section 7.2.2.1. The reason for invoking `setParentTasksSubDeadlines()` is because 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. The algorithm ends after processing all the tasks of the job.

7.2.2.1 Set Sub-deadlines of Parent Tasks Method

Algorithm 7.3 presents the algorithm for `setParentTasksSubDeadlines()` (abbreviated `setPTSubDL`). The input required by the method is a task t to process. The first step is to check if the execution time of task t is equal to 0, meaning it is a dummy task

(line 1). Recall from Section 7.1 that a dummy task is a task whose purpose is to only specify precedence relationships between the tasks of a job, and thus, dummy tasks have an execution time equal to zero. If the task to process *t* is a dummy task, the sub-deadline of each of *t*'s parent tasks is set to *t*'s own sub-deadline, and the method ends (see lines 2 to 5).

Otherwise, task *t* has an execution time greater than 0, and the method continues as follows. First, the highest sub-deadline that is found among all of *t*'s parent tasks is saved in the variable `highestSubDeadline` (line 7). Next, each of *t*'s parent tasks (denoted *pt*) is processed to see if their current sub-deadline should be updated and set to the value stored in `highestSubDeadline` (lines 8-15). The sub-deadline of each *pt* is only updated if the following two conditions are true: (1) the value stored in `highestSubDeadline` is larger than *pt*'s current sub-deadline (line 9) and (2) there is a single direct path from *pt* to *t*, that is there is a direct edge from *pt* to *t* and no other path from *pt* to *t* (see lines 10-13). To check condition (2), a method named `getListOfSucceedingTasksUnitl()`, which returns a list of tasks from *pt* (including *pt*) that form a path to *t*, is used. The return value of `getListOfSucceedingTasksUnitl()` is saved in a variable called `succeedingTasks` (line 10). If the number of tasks in `succeedingTasks` is equal to 1, it means that there is a direct path from *pt* to *t*, and *pt*'s sub-deadline is set to the value of `highestSubDeadline` (lines 11-13). On the other hand, if the number of tasks in `succeedingTasks` is greater than 1, the sub-deadline of the task is not changed and the next parent task of *t* is processed. After all of task *t*'s parent tasks are processed, the method returns (line 15).

An example of why condition (2) needs to be checked is shown in the DAG presented in Figure 7.3. The figure specifies the name of the tasks (t1 to t4) and the sub-

deadlines of each of the tasks (sd_t). Assume that the PD algorithm has already done the following: processed tasks t1 to t3, finished assigning a sub-deadline to t4, and just invoked the `setPTSubDL` method for t4. Since task t4 is not a dummy task, lines 1 to 6 are skipped and line 7 is executed, which assigns 21 to the variable `highestSubDeadline` because t3 has the highest sub-deadline among all of t4's parent tasks. If condition (2) (see lines 10-11) is not present in the `setPTSubDL` algorithm, task t1's sub-deadline would be set to 21. However, this would not make sense because it would cause task t1, which is a parent task of t2, to have a higher sub-deadline compared to t2. The reason that this situation can occur is because there are two paths from which task t1 can reach task t4: (1) a direct path from t1 to t4 and (2) a path from t1 through t2 and t3 to t4. Thus, this example of the precedence relationships between tasks, which has also been observed in some scientific workflows, demonstrates that the sub-deadline of a parent task should not be updated if there is more than one path from the parent task to the child task.

Algorithm 7.3: WFBB-RM algorithm's *setParentTasksSubDeadline()*

Input: task *t*
Output: none

```
1:  if t.getExecutionTime() = 0 then
2:      for each task pt in t's parent tasks list do
3:          call pt.setSubDeadline(t.getSubDeadline())
4:      end for
5:      return
6:  end if
7:  highestSubDeadline ← Get the highest sub-deadline among all of t's parent tasks.
8:  for each task pt in t's parent tasks list do
9:      if latestSubDeadline > pt.getSubDeadline() then
10:         call getListOfSucceedingTasksUnitl(pt, t) returning succeedingTasks
11:         if succeedingTasks.size() = 1 then
12:             pt.setSubDeadline(highestSubDeadline)
13:         end if
14:     end if
15: end for
```

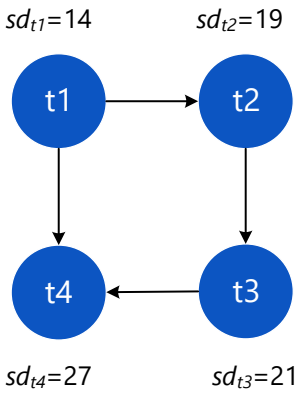

Figure 7.3. Sample DAG for illustrating the purpose of *getListOfSucceedingTasksUnitl()*.

7.2.3 Even Distribution of Job Laxity Algorithm

The Even Distribution of Job Laxity Algorithm (abbreviated ED) 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 7.1 that an execution phase in a multi-stage job comprises one or more tasks that can only start to execute after the previous execution phase, which also comprises a single task or set of tasks, is completed. 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.

The ED algorithm is presented in Algorithm 7.4 and a walkthrough of the algorithm is provided next. 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 determines the number of execution phases that the job contains (lines 2-6). This is accomplished by checking the execution phase attribute of each task *t* in job *j* (line 3) and adding task *t*'s execution phase to a list called `executionPhases`, if the task's execution phase is not already in the list (lines 4-5). The `executionPhases` list is then sorted in non-decreasing order (line 7). The next step of the algorithm is to calculate the amount of laxity

that each execution phase should be assigned (line 8). The laxity per each execution phase of a job j (denoted L_j^{ep}) is calculated as follows:

$$L_j^{ep} = {}^{L_j}\!/_{n_j^{ep}} \tag{7.6}$$

where L_j is the laxity of job j and n_j^{ep} is the number of execution phases in job j. 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 9 to 13. More specifically, the cumulative laxity of each execution phase ph for a job j is calculated as follows:

$$CL_j^{ph} = ph \times L_j^{ep} \tag{7.7}$$

where ph is an integer in the set $\{1, 2, 3, \ldots, n_j^{ep}\}$ that represents the execution phase. A map data structure named `cumulativeLaxities` is used to store the cumulative laxity for each execution phase (line 9). Note that the ED algorithm assigns each phase of execution the same amount of laxity (equal to L_j^{ep}). However, tasks that belong to an execution phase closer to the end of the job (i.e., an execution phase with a higher ph value) will have a higher maximum laxity value (equal to CL_j^{ph}) compared to tasks that belong to an execution phase that is near the start of the job (i.e., an execution phase with a smaller ph value). This is because the tasks belonging to an execution phase with a higher ph value can use the unused laxity from the tasks in the preceding execution phases.

The last phase of the algorithm (lines 14-27) uses the cumulative laxity values to calculate and assign a sub-deadline for each of job j's tasks. More specifically, the following operations on each task t of job j are performed. First, the execution phase of the task t is retrieved. If task t is a dummy task (recall the definition in Section 7.1), the execution phase of the task is obtained by finding the parent task of t that has the latest sample completion time (SCT_t) (see lines 15-17). Otherwise, the execution phase is

obtained directly from *t* as shown in line 18. After obtaining the execution phase of the task, the cumulative laxity of the task is retrieved from the `cumulativeLaxities` map using the value of the execution phase as the key (line 21). The sub-deadline of the task is then calculated using Eq. 7.5 (see Section 7.2.2) and assigned to the task (lines 22-23). Similar to the PD algorithm, the ED algorithm invokes the `setPTSubDL` method if the task *t* has more than 1 parent task. After all the tasks of job *j* are processed, the algorithm ends (line 27).

Algorithm 7.4: Even Distribution of Job Laxity Algorithm

Input: job *j*
Output: none

```
1:  jobLaxity ← j.getLaxity()
2:  Create an empty list named executionPhases.
3:  for each task t in job j do
4:      if executionPhases does not contain t's execution phase then
5:          executionPhases.add(t.getExecutionPhase())
6:  end for
7:  Sort executionPhases in non-decreasing order.
8:  laxPerEP ← jobLaxity / executionPhases.size()
9:  Create an empty map, cumulativeLaxities <execution phase, cumulative laxity>.
10: for i = 0 to (executionPhases.size() -1) do
11:     cl ← (i + 1) * laxPerEP
12:     cumulativeLaxitites.put(executionPhases.get(i), cl)
13: end for
14: for each task t in job j do
15:     if t.getExecutionTime() <= 0 then
16:         latestTask ← t.getLatestParentTask()
17:         ep ← latestTask.getExecutionPhase()
18:     else
19:         ep ← t. getExecutionPhase ()
20:     end if
21:     cumulativeLaxity ← cumlativeLaxities.get(ep)
22:     subDL ← t.getSCT() + cumulativeLaxity
23:     t.setSubDeadline(subDL)
24:     if t has more than one parent task then
25:         call setParentTasksSubDeadline (t)
26:     end if
27: end for
```

7.3 WFBB-RM Matchmaking and Scheduling Algorithm

This section describes the WFBB-RM technique'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 7.3.1) and (2) the *Job Remapping algorithm* (described in Section 7.3.2). 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 complete its execution before its deadline, the Job Remapping algorithm is called to remap job *j* and a set of jobs that may have caused *j* to miss its deadline.

7.3.1 Job Mapping Algorithm

The Job Mapping algorithm is comprised of two methods: (1) `mapJob()` presented in Algorithm 7.5 and (2) `mapJobHelper()` described in Algorithm 7.6. Note that the variables shown in the algorithms that are underlined indicate that the variables are fields belonging to the WFBB-RM algorithm instead of being local variables. A walkthrough of `mapJob()` is provided first, followed by the 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 parameter `tsp`, which specifies the *task scheduling policy*, these are the same input parameters as used by the DBW algorithm (described in the previous section). The method returns true if the job *j* can be 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, the WFBB-RM algorithm's `rootJob` field is set to *j* (line 2). The `rootJob` field stores the current job that is being mapped by the system. The third step is to clear the WFBB-RM

algorithm's `prevRemapAttempts` list (line 3), which stores the various sets of jobs that a job remapping attempt processes. The WFBB-RM algorithm'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, the WFBB-RM algorithm'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 start 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 7.6: `mapJobHelper()` (line 6).

Algorithm 7.5: WFBB-RM 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 earliest start time of task *t* is retrieved 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*. On the other hand, if an earliest start time for task *t* is determined (i.e., `getEarliestStartTime()` does not return -1) (line 6), `mapJobHelper()` continues to process task *t*, and the expected start time of *t* is calculated depending on the value of the WFBB-RM algorithm's `taskSchedulingPolicy` field, which is initialized by `mapJob()` (recall Algorithm 7.5). If `taskSchedulingPolicy` is set to TSP1, the expected start time of the task is not changed from the value obtained in line 5. However, if `taskSchedulingPolicy` is set to TSP2 (line 7), the expected start time of the task is set as shown in line 8. The completion time of the task is then calculated based on the expected start time of the task as shown in line 9.

Algorithm 7.6: WFBB-RM 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
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 28
19:                 else
20:                     Map t on r.
21:                     mappedTasks.add(t)
22:                 end if
23:             end if
24:         end if
25:     end for
26:     tasksToMap.removeAll(mappedTasks)
27: end while
28: if isJobMapped = true then
29:     mappedTasks.clear()
30:     mappedJobs.add(j)
31:     return true
32: else
33:     call mapJobHelper(job, true, false)
34:     return false
35: end if
```

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., if task *t* is a dummy task (defined in Section 7.1)) (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 WFBB-RM algorithm'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 start time on the system. 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 parameter `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 7.7: `remapJob()` (described in more detail in Section 7.3.2) 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 28 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 (line 29), job *j* is added to the WFBB-RM algorithm's `mappedJobs` list (line 30), and true is returned (line 31). Otherwise, `isJobMapped` is set to false, meaning job *j* cannot be scheduled to meet its deadline (line 32). This leads to `mapJobHelper()` being re-invoked but this time with the `checkDeadline` parameter set to false, which will map job *j* even if it misses its deadline (line 33). False is then returned (line 34) to indicate job *j* will not meet its deadline.

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 parameter `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 3-27. 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 28-31 are executed (as described earlier), and then the method returns.

7.3.2 Job Remapping Algorithm

The Job Remapping algorithm is comprised of two methods: (1) `remapJob()` presented in Algorithm 7.7 and (2) `remapJobHelper()` outlined in Algorithm 7.8. A discussion of `remapJob()` is provided first, followed by a discussion on `remapJobHelper()`. The input parameters required by `remapJob()` include a job *j* to remap and a Boolean `isRootJob`. The `isRootJob` parameter 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.

The first step of `remapJob()` is to set the WFBB-RM algorithm's `taskSchedulingPolicy` field to TSP1 so the tasks that are remapped are scheduled to execute at their earliest possible start times (line 1). The second step is to invoke Algorithm 7.8: `remapJobHelper()` (line 2). Recall from line 4 of Algorithm 7.5 (`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 (tighter deadline). If `remapJobHelper()` returns true, `remapJob()` also returns true (line 3). On the other hand, if `remapJobHelper()` returns false, `remapJob()` continues (line 4) by checking the supplied `isRootJob` parameter. 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 6). Otherwise, the method continues and the WFBB-RM algorithm's `jobComparator` field is changed to the *Job Laxity Comparator* (line 8) 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 9).

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:

$$NL_j = \frac{L_j}{SET_j} \tag{7.8}$$

where L_j is the laxity of job j and SET_j is the sample execution time of job j (recall Section 7.2). 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 and the equation $L_j = d_j - s_j - SET_j$ and Eq. 7.8, 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.

Algorithm 7.7: WFBB-RM algorithm's *remapJob()*

Input: job *j*, Boolean *isRootJob*
Output: a Boolean: true if job *j* and the set of jobs to remap are all 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
6:          return false
7:      end if
8:      Change jobComparator to the Job Laxity Comparator.
9:      return remapJobHelper(j, isRootJob)
10: end if
```

A walkthrough of `remapJobHelper()` (shown in Algorithm 7.8) is provided next. The input parameters 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 (line 1). This includes all the jobs in the WFBB-RM algorithm's `mappedJobs` list that have a scheduled start time or completion time 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 the WFBB-RM algorithm'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 the WFBB-RM algorithm'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 the `prevRemapAttempts` list (line 7). Next, the method checks if the supplied parameter `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 the WFBB-RM algorithm'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.

Algorithm 7.8: WFBB-RM algorithm's *remapJobHelper()*

Input: job *j*, Boolean *isRootJob*

Output: a Boolean: true if job *j* and the set of jobs to remap are all 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
```

The next step is to remove all the jobs in `jobsToRemap` from the system (line 11), which involves removing the jobs from the WFBB-RM algorithm'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 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 7.6: `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 7.6) with the `checkDeadline` input parameter 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).

7.4 Performance Evaluation of the WFBB-RM Technique

This section describes the simulation experiments conducted to evaluate the performance of the WFBB-RM technique (referred to simply as WFBB-RM). Two types of experiments are conducted to evaluate the effectiveness of WFBB-RM. The first type of experiments (see Section 7.5) are performed to investigate the effect of various system and workload parameters on the performance of WFBB-RM. 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 (see Section 7.6) are conducted to compare the performance of WFBB-RM with that of MRCP-RM (described in Chapter 4). MRCP-RM has objectives that are similar to that of WFBB-RM: minimizing the number of jobs that miss their deadlines when processing 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 7.4.1. Following this, a description of the system and workload parameters used in the factor-at-a-time experiments is provided in Section 7.4.2.

7.4.1 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. WFBB-RM and its associated algorithms are executed on the machine described. Similar to the previous performance evaluations described in this thesis, WFBB-RM is evaluated in terms of the following performance metrics in each simulation run:

- *Proportion of late jobs*, *P* (recall Section 4.4.1)
- *Average job turnaround time*, *T* (recall Section 4.4.1)
- *Average job matchmaking and scheduling time* (*O*) is the average processing time required by WFBB-RM to budget a job's deadline and match make and schedule a job. *O* is calculated as the total time required to process all the jobs during a simulation run divided by the total number of jobs arriving on the system in a simulation run.

O is a value that is measured using Java's `System.nanoTime()` [102] method, whereas *P* and *T* are values produced as output of the simulation. Similar to Section 4.4.1, the *O*-by-*T* ratio (denoted *O/T*) is used as an indicator of the processing overhead of WFBB-RM.

7.4.2 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, which come from various fields of study, 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 [113] and [114].

CyberShake is a seismology application that is created by the Southern California Earthquake Center to predict earthquake hazards in a region. More specifically, CyberShake uses the Probabilistic Seismic Hazard Analysis technique to identify all ruptures within 200 km of the site of interest. For each rupture, CyberShake calculates synthetic seismograms and then extracts the peak intensity measures from each seismogram. The peak intensity values are then combined with the original rupture probabilities to generate the probabilistic seismic hazard curves. The DAG of the CyberShake workflow is presented in Figure 7.4. The DAG shows that there are five phases of execution. The first, second, and fourth execution phases each contain multiple tasks to execute, whereas the third and fifth execution phases each only have one task to execute.

The Laser Interferometer Gravitational Wave Observatory (LIGO) Inspiral Analysis workflow is designed and used to search for and analyze gravitational waveforms in data collected by large-scale interferometers. An interferometer is an apparatus that uses the interference of waves to measure and analyze very small phenomena, such as small displacements (e.g., wavelength) and refractive index changes. The input data is partitioned into multiple blocks so that the data can be analyzed in parallel. Furthermore, the

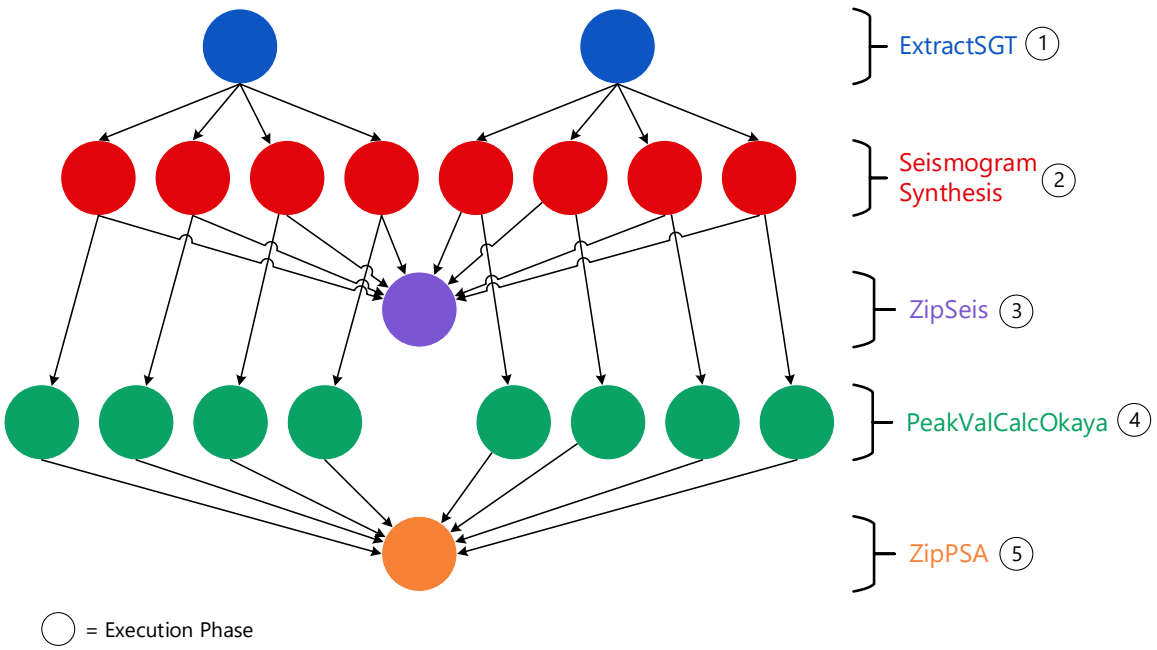

Figure 7.4. DAG of a sample CyberShake application [113].

application also generates a subset of waveforms (TmpltBank tasks) from each block of data to further facilitate parallel processing. Figure 7.5 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 to process (i.e., TmpltBank tasks). For example, in the sample LIGO workflow, Block 1 comprises 4 waveforms to process and Block 2 has 3 waveforms to process.

The Epigenomics (Genome) workflow is created by the University of Southern California Epigenome Center for automating several commonly used operations in genome sequence processing. The input of the workflow is DNA sequence data generated by the Illumina-Solexa Genetic Analyzer system that is partitioned into several pieces to facilitate parallel processing. Figure 7.6 presents the DAG of a sample Genome workflow, which is characterized by one or more *lanes*, each of which starts with the execution of a fastQSplit

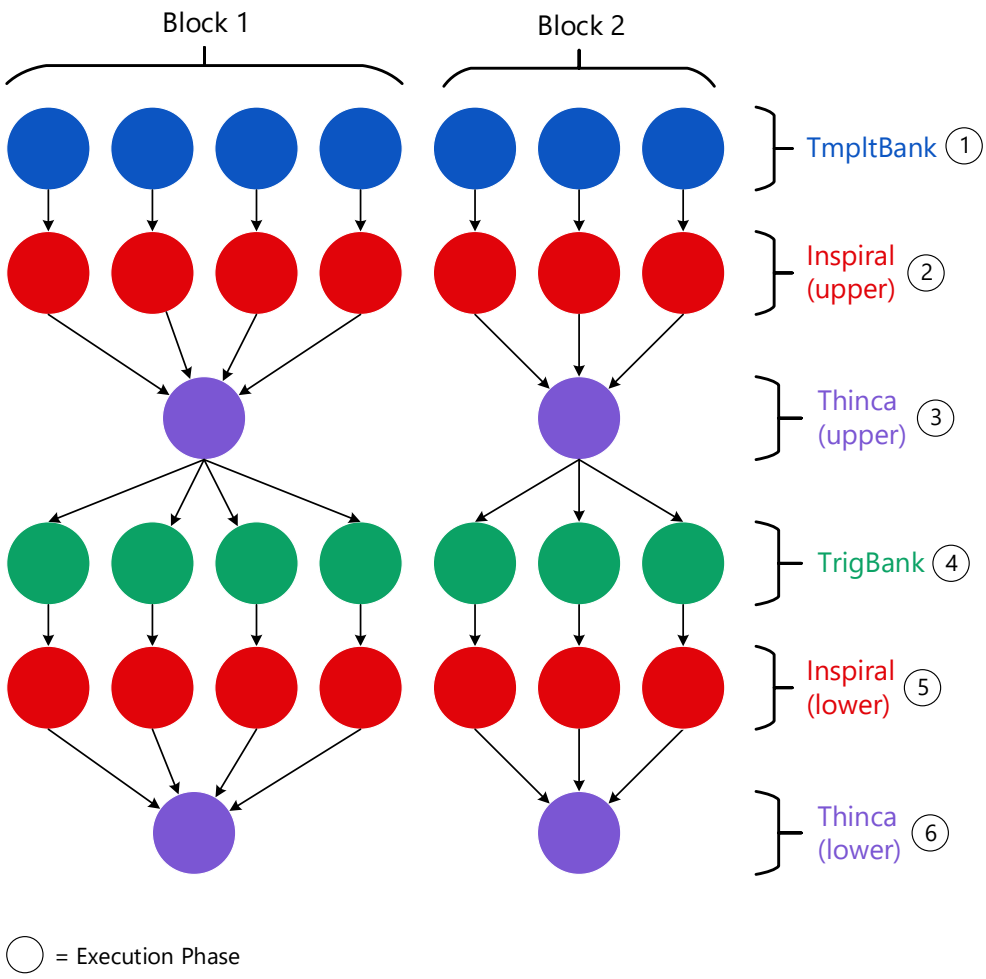

Figure 7.5. DAG of a sample LIGO Inspiral Analysis application [113].

task. The operations that are performed on the data involve the following: removing unnecessary (contaminating) sequences, mapping sequences into their correct locations according to the reference genome, and calculating the density of a sequence at each position in the genome. The Genome workflow has at least 8, but can have up to 9 phases of execution. If there is more than one lane in the workflow, as shown in the example in Figure 7.6, 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). On the other hand, if there is only one lane in the workflow, the global mapMerge stage is not needed, and thus there will only be 8 phases of execution.

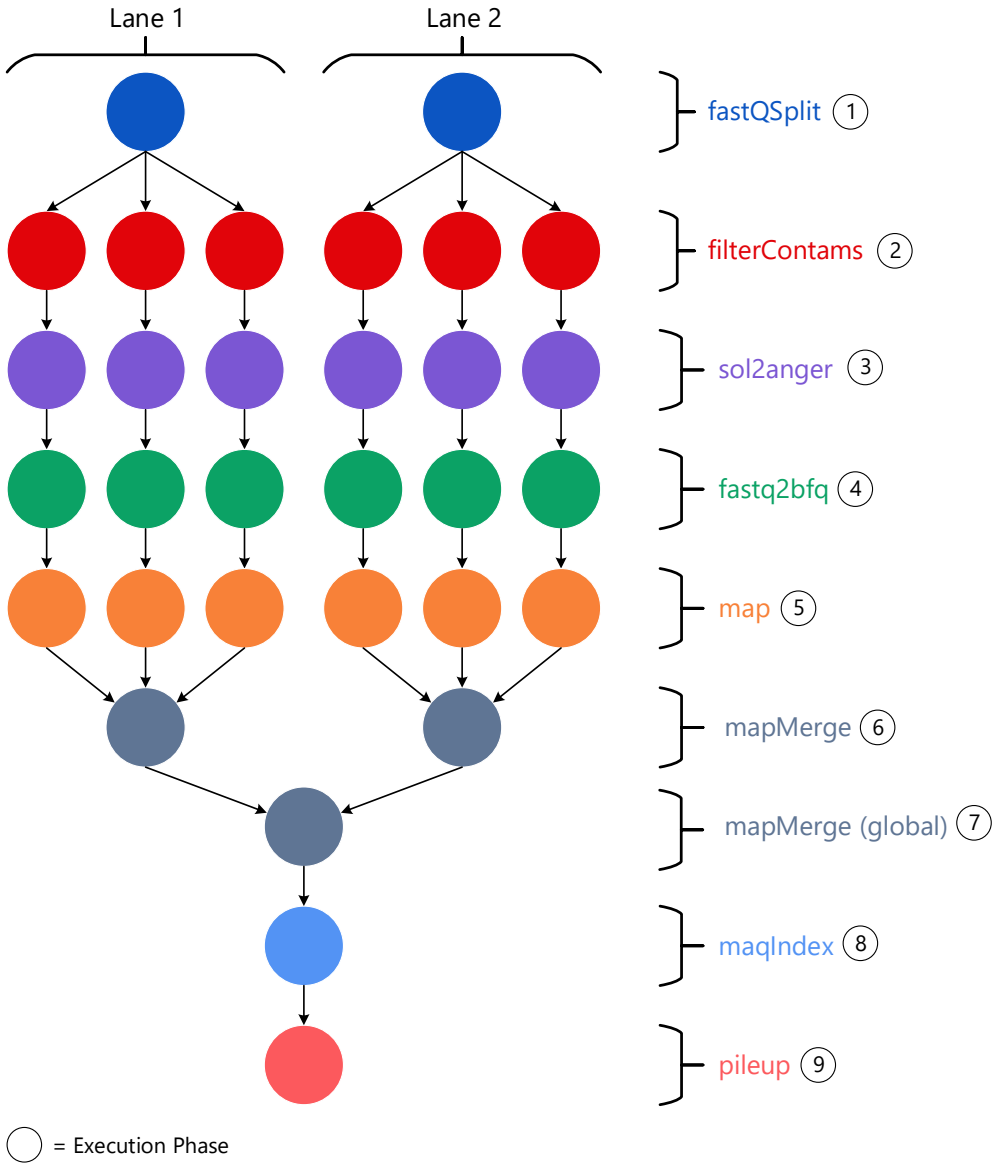

Figure 7.6. DAG of a sample Epigenomics application [113].

Table 7.1 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 7.1 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 the workload described in Section 4.4.3. 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: s*mall*, *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 [113]. 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 job, LIGO job, and Genome job on a single resource is equal to 1551 sec, 13300 sec, and 160213 sec, respectively. The parameters λ_{CS}, λ_{LG}, and λ_{GN} specify the 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 7.2) with an *execution time multiplier* (*em*) and adding the resulting value to s_j. The parameter *em* is used to determine the laxity of the 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.

Table 7.1. System and Workload Parameters for the WFBB-RM Factor-at-a-Time Experiments.

Parameter	***Values***	***Default Value***
Workload		
Type	{CyberShake, LIGO, Genome}	-
Job arrival rate (job/sec)	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 and $x \sim$Bernoulli (p), p = 0.5, and s_{max} = {1, 5, 25} * 10^4	s_{max}= 50000
Job Deadline, d_j (sec)	$d_j = \lceil s_j + SET_j^R * em \rceil$ 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 WFBB-RM		
Laxity Distribution Algorithm	{PD, ED}	-
Approach to calculate the job laxity	{SL, TL}	-
Task Scheduling Policy	{TSP1, TSP2}	-

Note: DU = discrete uniform distribution, *U* = uniform distribution

The remaining components of the table describe the system used to execute the jobs and the configuration of WFBB-RM. The number of resources (m), which represents the number of nodes in the distributed environment 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 7.1, c_r

specifies the number of tasks that a resource *r* can execute in parallel at any given point in time. The configuration of WFBB-RM is defined as *x*-*y*-*z* where *x* specifies the laxity distribution algorithm (i.e., PD or ED, described in Section 7.2), *y* specifies the approach to calculate the laxity of the job (i.e., SL or TL, described in Section 7.2), and *z* specifies the task scheduling policy (i.e., TSP1 or TSP2, described in Section 7.3.1). In total, there are 8 different WFBB-RM 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.

7.5 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* in most cases are observed to remain less than ±5% of the respective average values at a confidence level of 95%. 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 WFBB-RM 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 WFBB-RM configurations.

To provide clarity of presentation, only the results of the two WFBB-RM 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. More specifically, the two WFBB-RM 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

The complete results of the factor-at-a-time experiments (i.e., the results of all 8 WFBB-RM configurations for each of the three workloads) can be found in Appendix D.I to Appendix D.III.

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.

7.5.1 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 Figure 7.7 and Figure 7.8. 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 WFBB-RM 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}, WFBB-RM's Job Remapping algorithm, which is a source of overhead, is invoked more often, contributing to the increase in *O*.

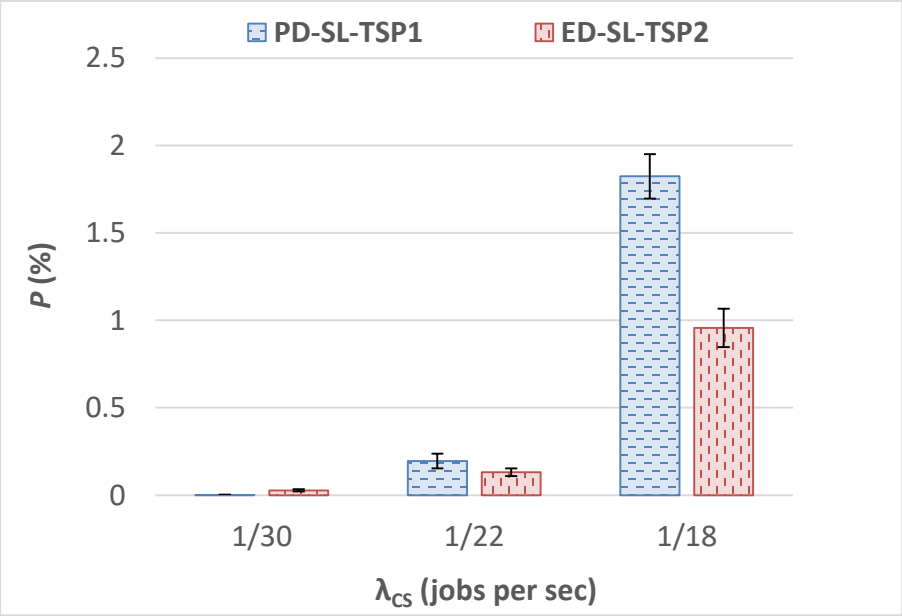

Figure 7.7. Effect of λ_{CS} on *P* when using the CyberShake workload.

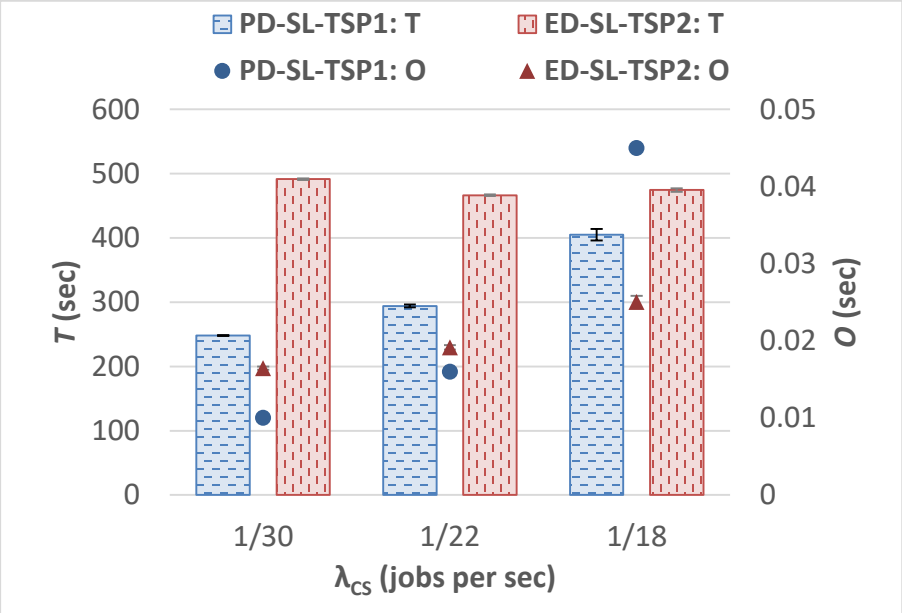

Figure 7.8. Effect of λ_{CS} on *T* and *O* when using the CyberShake workload.

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 Figure 7.8, 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 start times. However, when λ_{CS} is 1/18 jobs per sec, 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 7.2 and Table 7.3 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 WFBB-RM 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.

Table 7.2. LIGO workload: effect of λ_{LG} on *P*, *T*, and *O*.

λ_{LG} (jobs/sec)	***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 ±0.01	0.02 ±0.01	1346 ±0.6	1346 ±0.6	0.008 ±0.00	0.008 ±0.00
1/180	0.11 ±0.01	0.11 ±0.01	1466 ±4.6	1466 ±4.6	0.009 ±0.00	0.009 ±0.00
1/150	1.03 ±0.12	1.06 ±0.12	2005 ±29	2006 ±28	0.017 ±0.001	0.016 ±0.001

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 time, regardless of their respective sub-deadlines. Over all the experiments performed to investigate the effect of the job arrival rate, the results demonstrate that WFBB-RM can achieve low values of *P*

(less than 2% even at high arrival rates) and has a low processing overhead as indicated by the small *O* (less than 0.025 sec) and small *O/T* (less than 0.005%).

Table 7.3. Genome workload: effect of λ_{GN} on *P*, *T*, and *O*.

λ_{GN} (jobs/sec)	***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 ±0.00	0.01 ±0.00	17544 ±927	17544 ±927	0.008 ±0.000	0.008 ±0.000
1/2290	0.07 ±0.01	0.07 ±0.01	17963 ±1007	17963 ±1007	0.008 ±0.000	0.008 ±0.000
1/1800	1.43 ±0.45	1.40 ±0.44	52312 ±12915	52472 ±13003	0.048 ±0.015	0.051 ±0.016

7.5.2 Effect of Earliest Start Time of Jobs

The impact of the earliest start time of jobs on system performance is described in this section. Figure 7.9 and Figure 7.10 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

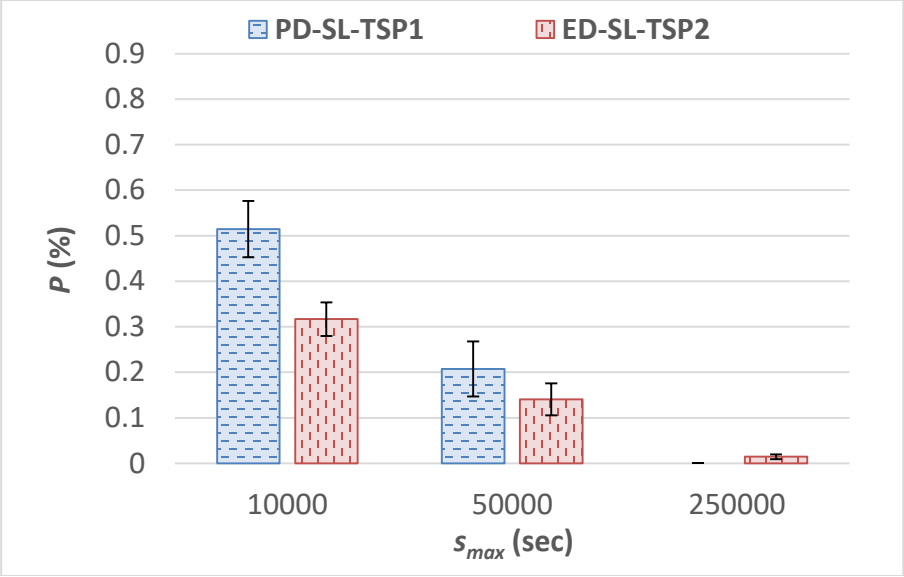

Figure 7.9. Effect of s_{max} on P when using the CyberShake workload.

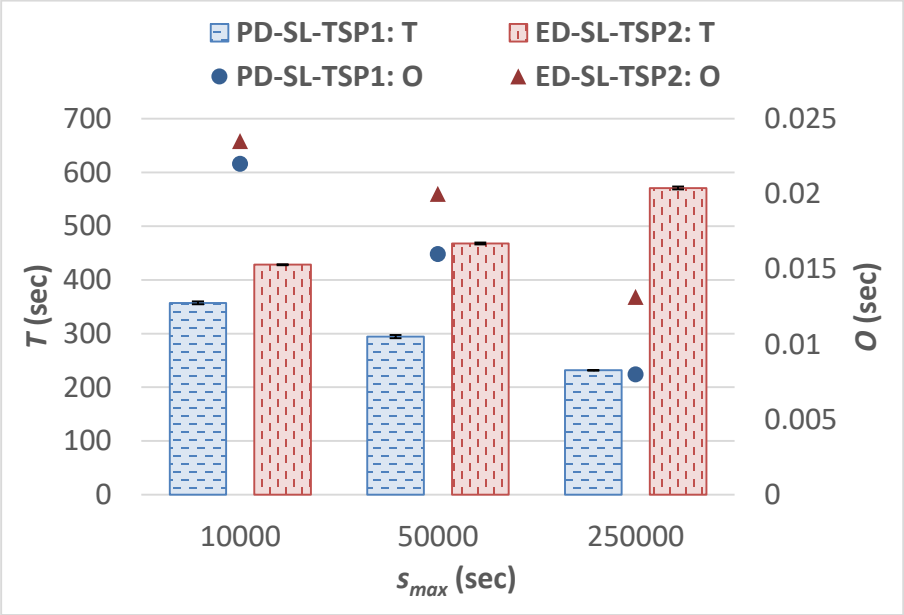

Figure 7.10. Effect of s_{max} on T and O when using the CyberShake workload.

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.

The results of the experiments using the LIGO workload are presented in Table 7.4. 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 Figure 7.9 and Figure 7.10 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 sec) is much smaller compared to that of the LIGO workload (13300 sec).

Table 7.4. 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**
10000	0.10 ±0.01	0.10 ±0.01	1450 ±3.3	1450 ±3.3	0.009 ±0.000	0.009 ±0.000
50000	0.11 ±0.01	0.11 ±0.01	1466 ±4.6	1466 ±4.6	0.009 ±0.000	0.009 ±0.000
250000	0.09 ±0.01	0.08 ±0.01	1441 ±4.7	1427 ±4.1	0.009 ±0.000	0.009 ±0.000

Table 7.5 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., 50000 and 250000 sec) 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 (~160213 sec 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 7.5. 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**
10000	0.04 ±0.01	0.04 ±0.01	17693 ±959	17693 ±959	0.008 ±0.000	0.008 ±0.000
50000	0.07 ±0.01	0.07 ±0.01	17963 ±1007	17963 ±1007	0.008 ±0.000	0.008 ±0.000
250000	0.08 ±0.01	0.08 ±0.02	18171 ±1049	18171 ±1049	0.008 ±0.000	0.008 ±0.000

7.5.3 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 Figure 7.11 and Figure 7.12, 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, WFBB-RM 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.

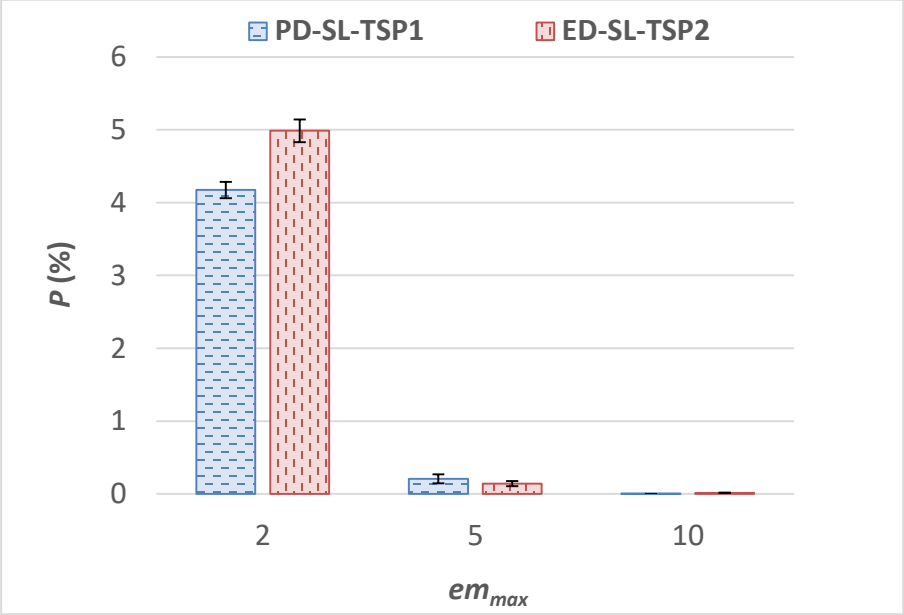

Figure 7.11. Effect of em_{max} on P when using the CyberShake workload.

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 ED-SL-TSP2.

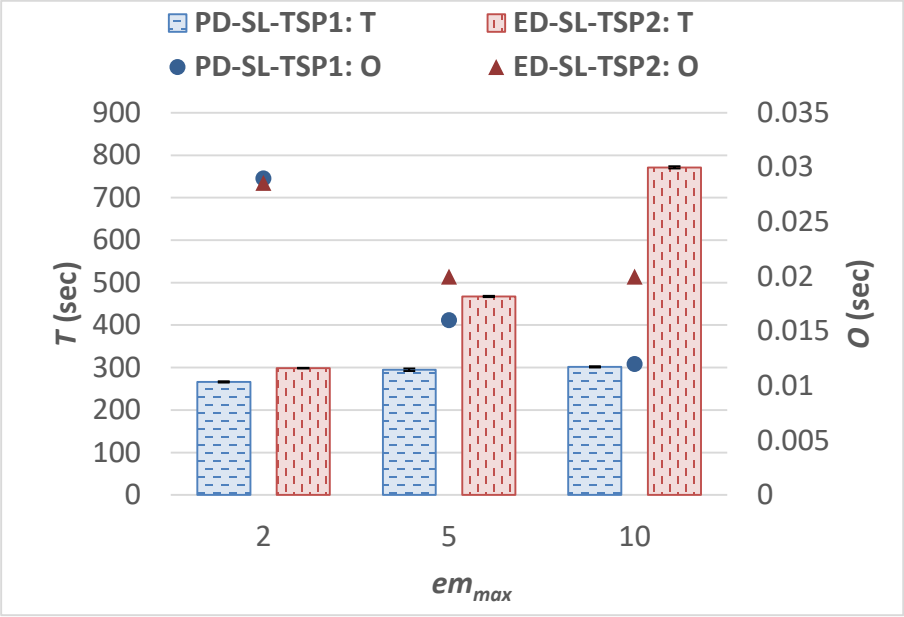

Figure 7.12. Effect of em_{max} on T and O when using the CyberShake workload.

The results of the experiments using the LIGO workload and Genome workload are presented in Table 7.6 and Table 7.7, 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 WFBB-RM 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 sec), and the processing overhead, as indicated by O/T, is less than 0.01% for all the experiments described in this sub-section.

Table 7.6. 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 ±0.14	2.43 ±0.14	1458 ±4.4	1457 ±4.4	0.012 ±0.000	0.011 ±0.000
5	0.11 ±0.01	0.11 ±0.01	1466 ±4.6	1466 ±4.6	0.009 ±0.000	0.009 ±0.000
10	0.04 ±0.01	0.04 ±0.01	1458 ±6.2	1463 ±4.6	0.009 ±0.000	0.008 ±0.000

Table 7.7. 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 ±0.12	0.49 ±0.12	17933 ±1001	17933 ±1001	0.009 ±0.000	0.009 ±0.000
5	0.07 ±0.01	0.07 ±0.01	17963 ±1007	17963 ±1007	0.008 ±0.000	0.008 ±0.000
10	0.03 ±0.01	0.03 ±0.01	17963 ±1007	17963 ±1007	0.007 ±0.000	0.007 ±0.000

7.5.4 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 Figure 7.13 and Figure 7.14), 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 the WFBB-RM's Job Remapping algorithm being invoked more often.

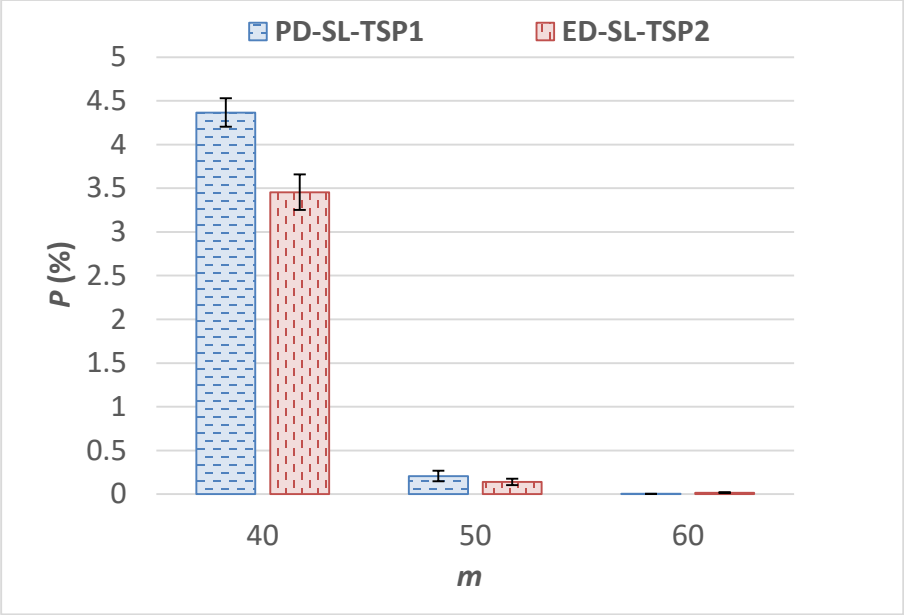

Figure 7.13. Effect of *m* on *P* when using the CyberShake workload.

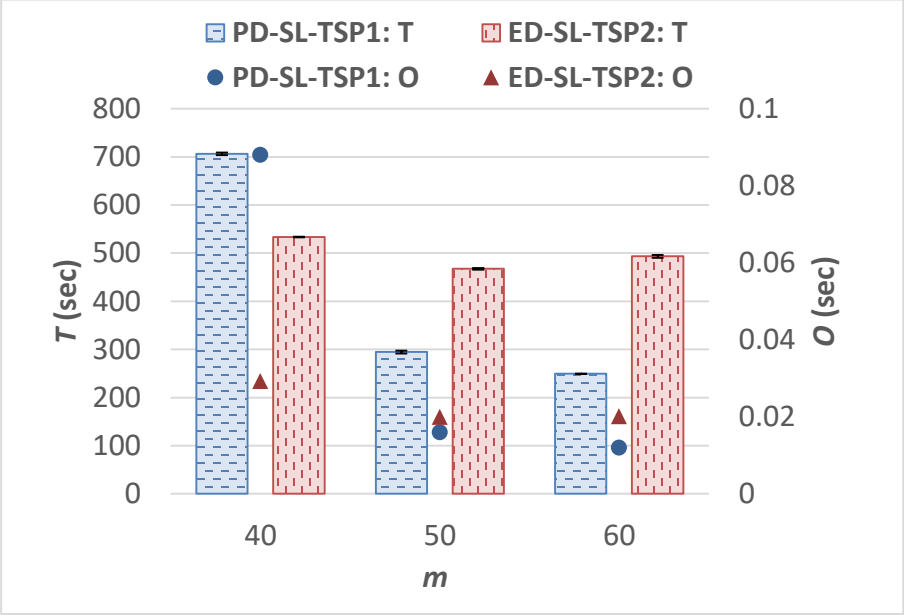

Figure 7.14. Effect of *m* on *T* and *O* when using the CyberShake workload.

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 there being more resources available in the system when *m* is 60, leading to a lower contention for resources and a smaller *P*, and thus the Job Remapping Algorithm, which remaps jobs to start executing at their earliest possible start times, does not need to be invoked as often. 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 overall, ED-SL-TSP2 achieves a smaller *P* and the most significant reduction in *P* is observed when *m* is 40 (see Figure 7.13). Similar to the results presented in the previous sections (see Figure 7.7, 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 effectively using the laxity of jobs to delay the execution of jobs with a later deadline to execute jobs with an earlier deadline. However, as shown in Figure 7.14, 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 to execute 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 Figure 7.13).

The results of the experiments using the LIGO workload (see Table 7.8) and the Genome workload (see Table 7.9) 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 start time. This in turn leads to a slight increase in *O*.

Table 7.8. 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 ±0.27	4.14 ±0.27	3210 ±125	3218 ±126	0.034 ±0.003	0.032 ±0.003
50	0.11 ±0.01	0.11 ±0.01	1466 ±4.6	1466 ±4.6	0.009 ±0.000	0.009 ±0.000
60	0.03 ±0.01	0.03 ±0.01	1360 ±1.1	1360 ±1.1	0.010 ±0.000	0.010 ±0.000

Table 7.9. 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 ±0.40	1.30 ±0.42	52320 ±13743	52106 ±13597	0.032 ±0.011	0.035 ±0.012
50	0.07 ±0.01	0.07 ±0.01	17963 ±1007	17963 ±1007	0.008 ±0.000	0.008 ±0.000
60	0.02 ±0.00	0.02 ±0.00	17583 ±935	17583 ±935	0.009 ±0.000	0.009 ±0.000

7.6 Comparison of WFBB-RM and MRCP-RM

This section discusses the results of the experiments conducted to compare the performance of WFBB-RM with that of MRCP-RM (described in Chapter 4). Recall that the MRCP-RM technique is only applicable to jobs with two phases of execution such as MapReduce jobs, whereas in addition to MapReduce jobs, WFBB-RM can also handle jobs with different structures and more than two execution phases. Thus, the workload that is used in this comparison is the Generic Synthetic MapReduce workload (described in Section 4.4.3) that is used in the experiments to evaluate the performance of MRCP-RM as described in Section 4.6. WFBB-RM is configured to use PD-SL-TSP1, which is observed to have the best performance when processing the Generic Synthetic MapReduce workload. Factor-at-a-time experiments are performed to investigate the effect of various system and workload parameters on the performance of WFBB-RM and MRCP-RM. The results of the experiments that show the effect of job arrival rate (λ) and the effect of the number of resources (m) on system performance are described in Section 7.6.1 and Section 7.6.2, respectively.

7.6.1 Effect of Job Arrival Rate

Figure 7.15 and Figure 7.16 present the performance of WFBB-RM and MRCP-RM in terms of P, T, and O as λ is varied. As shown in Figure 7.15, 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 both WFBB-RM and MRCP-RM have comparable values of P with MRCP-RM achieving a 6% lower 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 (on average 11% lower) *P* compared to that achieved by WFBB-RM. 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 WFBB-RM starts to outperform MRCP-RM. This can be attributed to the very high contention for resources (average resource utilization is approximately 0.95), leading to jobs queuing up on the system and MRCP-RM having to solve complex CP Models comprising a larger number of decision variables and constraints. MRCP-RM requires more time to solve these complex CP Models, 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 WFBB-RM achieves a significantly lower *O* compared to MRCP-RM (on average 85% lower) (see Figure 7.16).

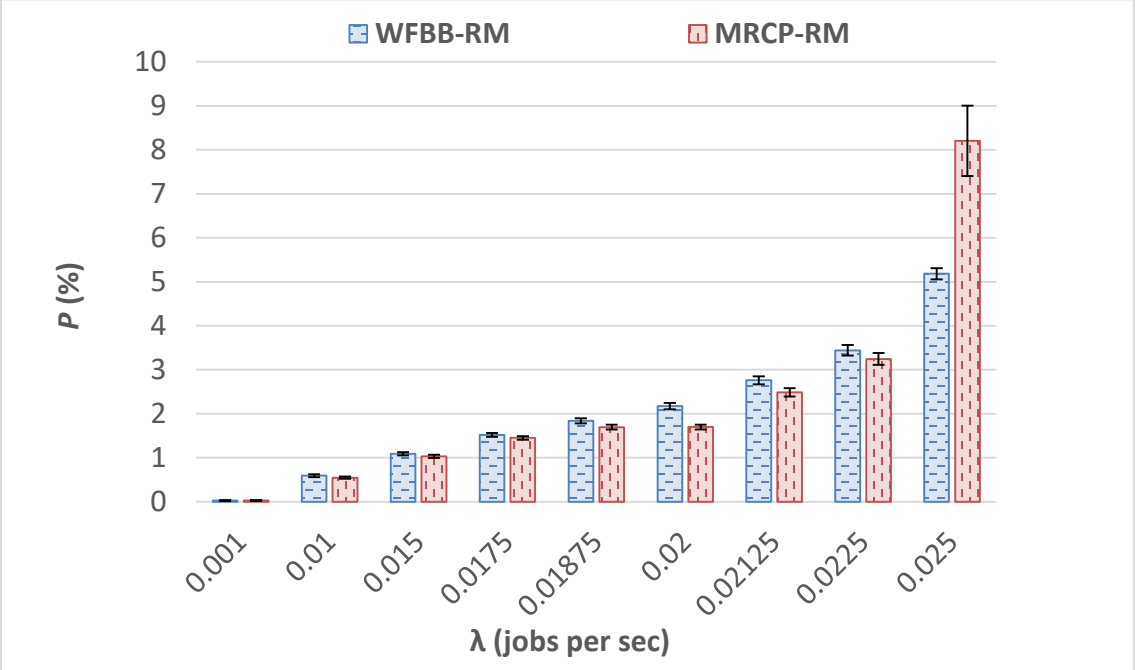

Figure 7.15. WFBB-RM vs MRCP-RM: effect of λ on *P*.

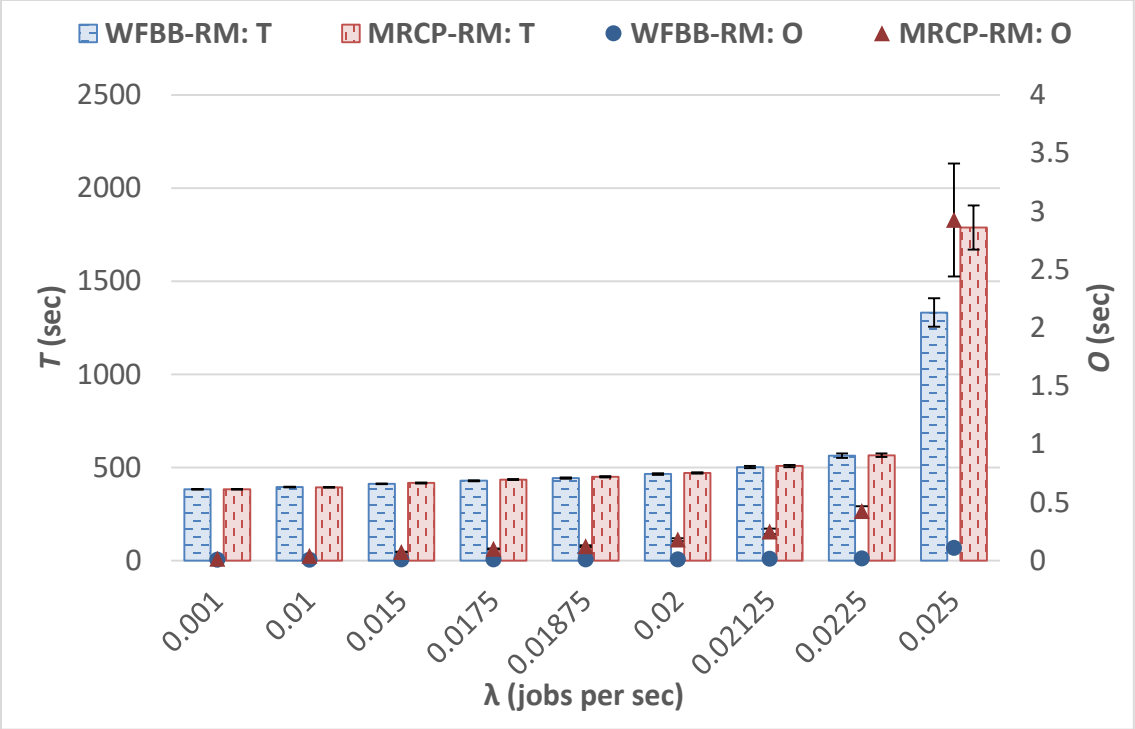

Figure 7.16. WFBB-RM vs MRCP-RM: effect of λ on *T* and *O*.

7.6.2 Effect of the Number of Resources

The performance of WFBB-RM and MRCP-RM in terms of *P*, *T*, and *O* when the number of resources (*m*) is varied are presented in Figure 7.17 and Figure 7.18. Similar to the results showing the effect of λ, when there is a low-to-moderate resource contention, such as when *m* is 100 or *m* is 50, it is observed that WFBB-RM and MRCP-RM perform comparably in terms of *P* and *T*. Furthermore, when *m* is 25, leading to a higher contention for resources (as reflected in an average resource utilization of approximately 0.8), it is observed that MRCP-RM achieves a 29% lower *P* compared to that achieved by WFBB-RM. From Figure 7.18, it is observed that MRCP-RM has a slightly higher *T* when *m* is 25. This can be attributed to MRCP-RM delaying the execution of jobs that have already missed their deadlines in favour of executing newly arriving jobs that have not missed their deadlines.

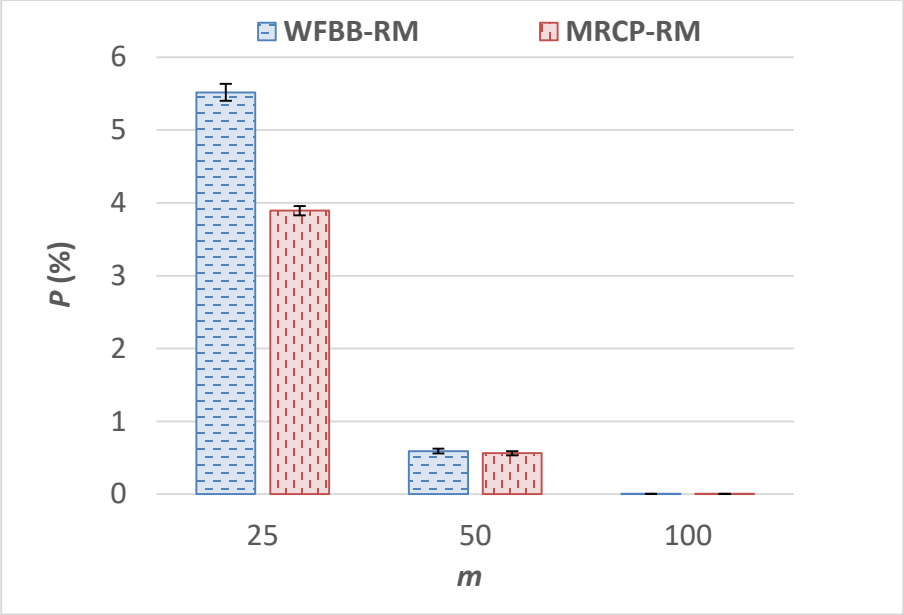

Figure 7.17. WFBB-RM vs MRCP-RM: effect of *m* on *P*.

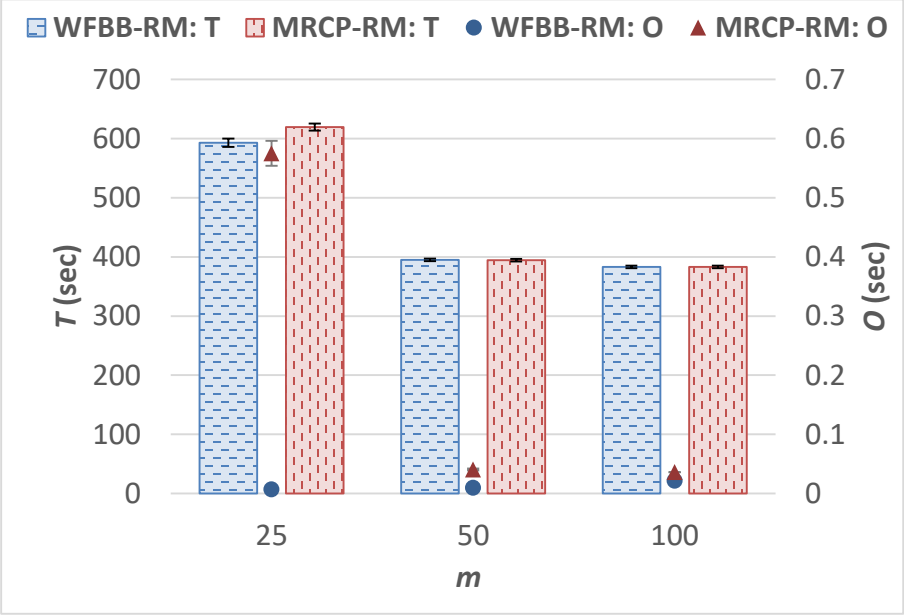

Figure 7.18. WFBB-RM vs MRCP-RM: effect of *m* on *T* and *O*.

Similar to the results described earlier, Figure 7.18 shows that WFBB-RM achieves a significantly lower *O* compared to MRCP-RM. Note that it is expected that generating a

higher contention for resources (by further reducing the value of *m* to be less than 25, for example) would lead to WFBB-RM outperforming MRCP-RM as observed when λ is 0.025 job per sec (refer to Figure 7.15).

The results of the other factor-at-a-time experiments that show the effect of task execution times, the effect of earliest start time of jobs, and the effect of job deadlines can be found in Appendix D.IV. Note that the results of these experiments shown in the appendix demonstrate a relative performance achieved by WFBB-RM and MRCP-RM that is similar to the results described in this section. MRCP-RM tends to achieve a lower *P* and has a similar *T* compared to WFBB-RM. However, WFBB-RM achieves a significantly lower *O*. Over all the experiments conducted to compare MRCP-RM and WFBB-RM, it is observed that MRCP-RM achieves up to a 29% lower *P*, but WFBB-RM achieves a 77% lower *O*. However, when the contention for resources is very high, such as when λ is 0.025 job per sec, WFBB-RM is observed to achieve a *P* that is 37% lower compared to the *P* achieved by MRCP-RM.

7.7 Summary and Discussion

This chapter describes a resource allocation and scheduling technique called WFBB-RM that can effectively and 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 WFBB-RM algorithm decomposes (budgets) the end-to-end deadline of a job into sub-deadlines, each of which is associated with a specific 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 WFBB-RM. 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 λ, or a decrease in s_{max}, or a decrease in em_{max}, or a decrease in m tends to lead to an increase in P due to the increased contention for resources.
- *WFBB-RM configuration using PD*: Overall, it is observed that using TSP1 generates lower or similar values of *P, T,* and *O* compared to TSP2. Furthermore, the two approaches used to calculate the laxity of the job (SL and 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 WFBB-RM configuration (in terms of *P*) for all three workloads experimented with is PD-SL-TSP1.
- *WFBB-RM configuration using 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 approach used to calculate the laxity of the jobs (SL and TL) achieve comparable performance.

- *PD vs ED*: 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 start times, regardless of their sub-deadlines.
- *Effectiveness of WFBB-RM*: For the system and workload parameters experimented with, it is observed that WFBB-RM 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 em_{max} is small, or m is small), P is less than 5% and on average 2.2% over all the experiments conducted.
- *Efficiency of WFBB-RM*: In all the experiments performed, WFBB-RM achieved low values of O (less than 0.05 sec and on average 0.02 sec). Furthermore, O/T, an indication of the matchmaking and scheduling overhead, is also very small (less than 0.01%) for all the experiments conducted.
- *Comparison with MRCP-RM*: A summary of observations resulting from the performance evaluation to compare WFBB-RM 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 or $m \geq 50$), WFBB-RM and MRCP-RM both achieve comparable performance in terms of P. When the contention for resources is moderately high (e.g., average resource utilization is approximately 0.8),

resulting in a λ of 0.02 jobs per sec or m of 25, for example, MRCP-RM outperforms WFBB-RM and achieves up to 29% lower P. At very high contention for resources (e.g., $\lambda \geq 0.025$ jobs per sec), WFBB-RM outperforms MRCP-RM.

Overall, the results of the experiments demonstrate that the objective of the research presented in this chapter that concerns the devising of an efficient resource allocation and scheduling technique for processing an open-stream of multi-stage jobs with SLAs on a distributed computing environment has been realized. WFBB-RM 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 WFBB-RM configuration to use is dependent on the workload to process; however, a good starting point is to use PD-SL-TSP1, followed by using ED-SL-TSP2. 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 use TSP2.

An interesting direction for future work is the investigation of new deadline budgeting algorithms for distributing the laxity of a multi-stage job among the job's constituent tasks that are not only based on the execution time of the tasks, but also based on additional attributes of the components of the DAG, which is used to model the multi-stage job. These attributes include the number of children belonging to tasks, the height of a task in the DAG, and whether the task is on the critical path of the DAG.

Chapter 8 Summary and Conclusions

Effective matchmaking and scheduling (resource management) techniques are crucial for harnessing the power of the underlying resource pool of a cloud or cluster and is required to attain high system performance (e.g., high job throughput and low job response times), satisfy QoS requirements of users as captured by SLAs, and maintain high resource utilization. The objective of this thesis is to devise effective resource management techniques for efficiently processing an open stream of multi-stage jobs (such as MapReduce type applications) with SLAs on a computing environment with a fixed number of resources, such as a private cluster or a set of resources acquired a priori from a public cloud. Each job submitted to the system is characterized by a SLA that includes an earliest start time, an execution time, and an end-to-end deadline. Multi-stage jobs require service from multiple system resources and are characterized by multiple phases of execution, where each phase of execution can comprise of one or more tasks to execute.

This thesis presents resource management techniques for processing both MapReduce type jobs (characterized by two phases of execution) and workflows with different types of precedence relationships and more than two phases of execution, including scientific workflows used in the domain of physics and biology. MapReduce/Hadoop has emerged as a popular technique and tool for performing Big Data analytics that includes analyzing data for making meaningful decisions in various types of environments, such as enterprise and scientific applications, and cyber-physical systems (e.g., sensor-equipped bridges, smart buildings, and industrial machinery). A key goal of this thesis is to devise resource management techniques that achieve high system performance as reflected in a low proportion of jobs missing their deadlines, while ensuring

the processing overhead is low. Conclusions derived from this thesis research and a summary of the key contributions of this thesis are presented in the upcoming sub-sections, followed by a discussion on directions for future work. Overall, as captured in the following discussion, the objective and goal of the thesis of devising matchmaking and scheduling techniques for efficiently processing an open stream of multi-stage jobs with SLAs have been achieved.

8.1 Resource Management Techniques for Processing a Batch of MapReduce Jobs with SLAs

In Chapter 3, resource management techniques for processing a batch of MapReduce jobs with SLAs are presented. The techniques formulate and solve the resource management problem as an optimization problem using two methods: (1) mixed integer linear programming (MILP) and (2) constraint programming (CP). The two formulations are implemented and solved using various commercial-off-the-shelf and open source software packages, leading to three approaches being devised. For all three approaches, the main objective is to minimize the number of jobs that miss their deadlines. A rigorous simulation-based performance evaluation of the three approaches is conducted using several batch workloads (see Section 3.7). The following insights are derived from the results of the experiments:

- *Superiority of Approach 3*: The results of the experiments showed that Approach 3, which implements and solves the CP Model using IBM CPLEX, achieves the lowest processing time overhead (*PO*). However, it also generated a schedule that produces a slightly higher batch workload completion time (*C*) compared to the

other two approaches. In addition, Approach 3 is the only approach able to process the large workloads comprising over 1000 tasks (see Section 3.7.2).

- Approach 1 and Approach 2 each have a case where they can generate a schedule that has the lowest *C* for the small workloads; however, the *PO* in these cases is much higher compared to the *PO* achieved by Approach 3.

8.2 MapReduce Constraint Programming based Resource Management Technique

A resource management technique based on constraint programming for processing an open stream of MapReduce jobs with SLAs, referred to as MRCP-RM, is presented in Chapter 4. MRCP-RM uses constraint programming because the results of the experiments described in Chapter 3 demonstrated the superiority of Approach 3, which solves the CP Model using IBM CPLEX, in being able to process workloads comprising over 1000 tasks, while incurring a low processing overhead. A number of simulation experiments are conducted using two synthetic MapReduce workloads to evaluate the effectiveness of the MRCP-RM technique. A summary of the results of these experiments is presented next.

- *Comparison with MinEDF-WC* [70] (see Section 4.5): The results show that MRCP-RM achieves a significantly lower proportion of late jobs, *P* (up to 93% lower) and a comparable or slightly lower average job turnaround time (*T*).
- *Effectively controlling P*: For most of the system and workload parameters experimented with (see Section 4.6), MRCP-RM achieves a *P* of less than 0.6%. In the scenarios where jobs are more susceptible to miss their deadlines and contention for resources is high, including workloads for which the execution time multiplier (*em*) is small, or number of resources in the system (*m*) is small,

or job arrival rate (λ) is high, or maximum map task execution time (me_{max}) is high, *P* is still observed to be low: 3.46%, 3.89%, 1.7%, and 1.96%, respectively.

- *Efficiency* and *scalability* (see Section 4.6): The average job matchmaking and scheduling time (*O*) is observed to increase when the contention for resources is high (e.g., high λ or *m* is small). However, the values of *O* are still observed to be quite low. For example, the highest *O* (equal to 0.57 sec) is observed when *m* is small (25), resulting in a high contention for resources. In addition, *O/T*, which is an indicator of the matchmaking and scheduling overhead, is observed to be less than 0.09% in all the factor-at-a-time experiments conducted, demonstrating that the matchmaking and scheduling overhead is small and MRCP-RM is thus scalable over a wide range of system and workload parameters experimented with. It is expected that for a reasonable contention for resources, MRCP-RM can work efficiently and achieve a reasonable *O* and *O/T*.

8.3 Hadoop Constraint Programming based Resource Management Technique

The focus of Chapter 5 is on describing the data-locality-aware Hadoop Constraint Programming based Resource Management technique, referred to as HCP-RM. HCP-RM adapts MRCP-RM so that it can be used on a real MapReduce system called Hadoop [25]. More specifically, the HCP-RM algorithm is implemented in a new scheduler for Hadoop called the CP-Scheduler. An in-depth prototyping and measurement based performance evaluation of HCP-RM (CP-Scheduler) is conducted on a Hadoop cluster deployed on Amazon EC2 using both a synthetic workload and a real workload. The performance of HCP-RM (CP-Scheduler) is compared to that of an Earliest Deadline First Hadoop

Scheduler (EDF-Scheduler), which is an extension of Hadoop's default FIFO scheduler implemented in this research to support job deadlines. The insights into system behaviour and performance gained from analyzing the results of the experiments are summarized next.

- *Superiority of HCP-RM over the EDF-Scheduler*: Over all the experiments conducted (see Section 5.6), HCP-RM generated a schedule that leads to a lower or equal P (on average 60% lower) and a lower T (on average 59% lower) compared to the EDF-Scheduler. This demonstrates the effectiveness of HCP-RM.
- *Small processing overhead*: The performance improvement of HCP-RM in terms of P and T over the EDF-Scheduler is accompanied by a higher O. However, O/T is still observed to be small (less than 0.92%), demonstrating the efficiency of HCP-RM.
- *Effect of error in execution times* (see Section 5.7): The investigation of error in user-estimated execution times showed that overestimated execution times lead to a lower P, comparable T, and slightly higher O compared to the case where there is no error in execution times. Conversely, underestimated execution times lead to lower performance in terms of P and T, but gives rise to a lower O compared to the case where there is no error in execution times.

8.4 Techniques for Handling Error in User-estimated Execution Times

In Chapter 6, techniques for handling error in user-estimated execution times (submitted as part of the SLA of the job) are discussed. A Prescheduling Error Handling (PSEH) technique that adjust the user-estimated execution times of jobs to make them more

accurate before the jobs are mapped by the resource management algorithm is presented. A rigorous performance evaluation of the PSEH technique is conducted on a Hadoop cluster deployed on Amazon EC2. More specifically, the performance of HCP-RM-EH that uses the PSEH technique is compared with the performance achieved by the original version of HCP-RM. Three models are used to generate the error in the user-estimated execution times. A summary of the results of the experiments, including the insights gained, is described next.

- *Effectiveness of the PSEH technique*: Overall, in the experiments conducted using the Constant Error Model (see Section 6.3.1), HCP-RM-EH is observed to achieve up to 50% lower P (on average 29% lower) compared to that achieved by HCP-RM. Furthermore, HCP-RM-EH achieves a P of 0 when f is 2 and λ is 1/30 jobs per sec or lower.
    - In the experiments using Feitelson's Error Model (see Section 6.3.2), both HCP-RM-EH and HCP-RM achieve the same values of P of less than 0.2% due to Feitelson's error model generating jobs with highly overestimated execution times. However, HCP-RM-EH achieves a 72% lower O.
    - When using the Variable Error Model (see Section 6.3.3), the results of the experiments show that HCP-RM-EH achieves a 54% lower P compared to that achieved by HCP-RM. This demonstrates that HCP-RM-EH is still effective when the error in execution times is not constant.
- The superior performance of HCP-RM-EH can be attributed to the PSEH technique being able to adjust the user-estimated execution times to make them more accurate. This enables HCP-RM-EH to make intelligent matchmaking and

scheduling decisions that lead to HCP-RM-EH achieving values of P that are lower compared to the values of P achieved by HCP-RM.

8.5 Workflow Budget-Based Resource Management Technique

A resource management technique, referred to as WFBB-RM, for processing an open stream of multi-stage jobs with SLAs is presented in Chapter 7. WFBB-RM decomposes (budgets) the end-to-end deadline of a job into sub-deadlines, each of which is associated with a specific 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. A rigorous simulation-based performance evaluation of WFBB-RM using workloads based on real scientific applications (workflows) are conducted. A summary of the insights gained into system behaviour and performance from the results of the experiments is provided next.

- *Effectiveness of WFBB-RM*: For the system and workload parameters experimented with (see Section 7.5), it is observed that WFBB-RM 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 em_{max} is small, or m is small), P is less than 5% and on average 2.2% over all the experiments conducted.
- *Efficiency of WFBB-RM*: In all the experiments performed (see Section 7.5), WFBB-RM achieved low values of O (less than 0.05 sec and on average 0.02 sec). Furthermore, O/T is also very small (less than 0.01%) over all the experiments conducted.

- *Comparison with MRCP-RM* (see Section 7.6): With an open stream of MapReduce jobs, when the contention for resources is low-to-moderate (e.g., average resource utilization is approximately less than 0.7), WFBB-RM and MRCP-RM achieve comparable values of *P*. At moderate-to-high resource contention (e.g., an average resource utilization of approximately 0.7 to 0.85), MRCP-RM achieves up to a 29% lower *P*. When the contention for resources is very high (e.g., average resource utilization is approximately 0.95), WFBB-RM starts to outperform MRCP-RM in terms of *P*. Over all the experiments conducted to compare MRCP-RM and WFBB-RM, it is observed that WFBB-RM achieves an *O* that is on average 85% lower compared to the *O* achieved by MRCP-RM.

8.6 Future Work

This section describes directions for future work and research. In line with previous work, this thesis focuses on an important class of systems in which MapReduce jobs are associated with deadlines but not with explicit priorities. Extending the MRCP-RM and HCP-RM algorithms to handle MapReduce jobs with deadlines as well as priorities forms an interesting direction for future research. On such a system, the resource manager may improve the performance of a higher priority job at the cost of a lower priority job if needed. A high-level approach for achieving this is briefly described next. With a workload comprising jobs with priorities, the objective function of the CP Model will be changed to minimize $\sum_{j \in J}(N_j * Pr_j)$ where Pr_j is the priority of job j. To lower this sum, the CPLEX solver will tend to favour meeting the deadlines of higher priority jobs (higher Pr_j) at the cost of missing the deadlines for low priority jobs. Moreover, as described in Section 6.4.1, devising a runtime error handling technique for dealing with the situation where jobs have

already started executing and their execution times are inaccurate, is also worthy of further investigation.

An interesting direction for future research concerns the modification of the resource management techniques for supporting *approximate (or partial) computations* [115], where an application is permitted to generate an approximate result that is less accurate or of poorer (but still acceptable) quality compared to the result produced by the full computation, when the application cannot complete the full computation before its deadline. The idea is to be able to return a result before the application's deadline instead of not returning any results or a result that is late. These types of applications, which can include numerical computation, statistical estimation, as well as video and voice transmission/processing applications [116], are typically characterized by a *mandatory component* and an *optional component* [116]. The application is considered to be completed if the mandatory component is completed. The optional component of the application enhances the quality (or accuracy) of the computation generated by the application and can either be fully completed, partially completed, or not executed at all. Ideally, both the mandatory and optional components of the application should be completed before the deadline of the application.

Another direction for future work is to adapt the resource management techniques to work in a distributed computing environment where the number of resources in the system can be dynamically increased or decreased. 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 whether one phase of execution needs to share 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.

The cost-performance trade-off of provisioning resources that have different levels of processing/network capabilities from the cloud can also be investigated. For example, a compute resource with a faster CPU and more memory maybe able to execute a job faster, but it will cost more for the user to provision the resource. Furthermore, the resource management techniques can also consider the trade-off between the cost associated with providing elasticity (or auto-scaling) as well as the cost (fine) associated with violating an SLA of a job (e.g., missing the deadline of the job due to scarcity of resources). Resource management techniques that minimize the overall cost incurred by the service provider form an interesting direction for future research.

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility", *Future Generation Computer Systems,* vol. 25, no. 6, June 2009, pp. 599-616.

[2] L. Columbus, "Roundup Of Cloud Computing Forecasts And Market Estimates, 2015", *Forbes*, 24 Jan. 2015. [Online]. Available: http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/ [Accessed: February 10, 2016]

[3] Gartner, "Gartner Says Worldwide Cloud Infrastructure-as-a-Service Spending to Grow 32.8 Percent in 2015", 18 May 2015. [Online]. Available: http://www.gartner.com/newsroom/id/3055225 [Accessed: February 10, 2016]

[4] R. Cohen, "Gartner Announces 2012 Magic Quadrant for Cloud Infrastructure as a Service", *Forbes*. Available: http://www.forbes.com/sites/reuvencohen/2012/10/22/gartner-announces-2012-magic-quadrant-for-cloud-infrastructure-as-a-service/ [Accessed: February 10, 2016]

[5] F. Gens, "IT Model in the Cloud Computing Era", *IDC Enterprise Panel*, August 2008.

[6] S. S. Manvi and G. K. Shyam, "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey", *Journal of Network and Computing Applications*, vol. 41, October 2013, pp. 424-440.

[7] F. Gens, "IT Cloud Services Forecast – 2008, 2012: A Key Driver of New Growth", *IDC Exchange*, October 2008.

[8] Amazon, "Amazon Elastic Cloud". [Online]. Available: http://aws.amazon.com/ec2 [Accessed: February 16, 2016].

[9] Microsoft, "Windows Azure". [Online]. Available: http://www.windowsazure.com/en-us/ [Accessed: February 16, 2016].

[10] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *International Journal of Supercomputer Applications*, vol.15, no.3, 2001, pp. 200-222.

[11] R. Buyya, S.K. Garg, and R.N. Calheiros, "SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions", *International Conference on Cloud and Service Computing (CSC)*, Hong Kong, China, 12-14 Dec. 2011, pp.1-10.

[12] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters”, *International Symposium on Operating System Design and Implementation (OSDI),* San Francisco, CA, USA, 6-8 December 2004, pp. 137–150.

[13] R. Bosch and M. Trick, “Integer programming”, *Search Methodologies*. Springer US, 2005, pp. 69-95.

[14] F. Rossi, P. Beek, and T. Walsh, “Chapter 4: Constraint Programming. *Handbook of Knowledge Representation*”, 2008, pp. 181-211.

[15] IBM. IBM ILOG CPLEX Optimization Studio. [Online]. Available: http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r5/index.jsp [Accessed: February 16, 2016].

[16] C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely, “Are User Runtime Estimates Inherently Inaccurate?”, in *Job Scheduling Strategies for Parallel Processing*, vol. 3277, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 253–263.

[17] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “Backfilling Using System-Generated Predictions Rather than User Runtime Estimates”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, 2007, pp. 789–803.

[18] W. Tang, N. Desai, D. Buettner, and Z. Lan, “Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P”, *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, USA, 19-23 April 2010, pp. 1–11.

[19] N. Lim, S. Majumdar, and P. Ashwood-Smith, “Resource Management Techniques for Handling Requests with Service Level Agreements”, *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Monterey, CA, USA, 6-10 July 2014, pp. 618 -625.

[20] N. Lim, S. Majumdar, and P. Ashwood-Smith, “Engineering Resource Management Middleware for Optimizing the Performance of Clouds Processing MapReduce Jobs with Deadlines”, *International Conference on Performance Engineering (ICPE)*, Dublin, Ireland, 24 -26 March 2014, pp.161-172.

[21] N. Lim, S. Majumdar, and P. Ashwood-Smith, “A Constraint Programming-Based Resource Management Technique for Processing MapReduce Jobs with SLAs on Clouds”, *International Conference on Parallel Processing (ICPP)*, Minneapolis, MN, USA, 9-12 Sept 2014, pp. 411-421.

[22] N. Lim, S. Majumdar, and P. Ashwood-Smith, “A Constraint Programming Based Hadoop Scheduler for Handling MapReduce Jobs with Deadlines on Clouds”, *International Conference on Performance Engineering (ICPE)*, Austin, TX, USA, 31 Jan – 4 Feb 2015, pp. 111-122.

[23] N. Lim and S. Majumdar, "Resource Management for MapReduce Jobs Performing Big Data Analytics", in *Big Data Management, Architecture, and Processing*, K.-C. Li, H. Jiang, and A. Zomaya, Eds. USA: CRC Press, Taylor & Francis Group, August 2016 (accepted for publication).

[24] N. Lim, S. Majumdar, and P. Ashwood-Smith, "MRCP-RM: a Technique for Resource Allocation and Scheduling of MapReduce Jobs with Deadlines", *IEEE Transactions on Parallel and Distributed Systems*, October 2016 (accepted for publication).

[25] The Apache Software Foundation, "Hadoop". [Online]. Available: http://hadoop.apache.org [Accessed: February 16, 2016].

[26] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini, "QoS–Aware Clouds", *International Conference on Cloud Computing (CLOUD)*, Miami, Florida, USA, 5-10 July 2010, pp. 321-328.

[27] P. Xiong, Y. Chi, S. Zhu, H.J. Moon, C. Pu and H. Hacigumus, "Intelligent management of virtualized resources for database systems in cloud environment", *International Conference on Data Engineering (ICDE)*, Hannover, Germany, 11-16 April 2011, pp. 87-98.

[28] Y. Yuan and W-C. Liu, "Efficient resource management for cloud computing", *International Conference on System Science, Engineering Design and Manufacturing Informatization (ICSEM)*, Guiyang, China, 22-23 Oct. 2011, pp. 233-236.

[29] C. Castillo, G.N. Rouskas, and K. Harfoush, "Resource co-allocation for large-scale distributed environments." *International Symposium on High performance distributed computing (HPDC)*, Munich, Germany, 11-13 June 2009*,* pp. 131-140.

[30] X. Wang, H. Xie, R. Wang, Z. Du, and L. Jin, "Design and implementation of adaptive resource co-allocation approaches for cloud service environments", *International Conference on Advanced Computer Theory and Engineering (ICACTE)*, Chengdu, China, 20-22 Aug. 2010, pp. V2-484-V2-488.

[31] R. Aoun, E.A. Doumith, and M. Gagnaire, "Resource Provisioning for Enriched Services in Cloud Environment", *International Conference on Cloud Computing Technology and Science (CloudCom)*, Indianapolis, USA, Nov. 30-Dec. 3 2010, pp. 296-303.

[32] V.C. Emeakaroha, I. Brandic, M. Maurer, and I. Breskovic, "SLA-Aware Application Deployment and Resource Allocation in Clouds", *International Conference on Computer Software and Applications Conference Workshops (COMPSACW)*, Munich, Germany, 18-21 July 2011, pp. 298-303.

[33] H. Goudarzi and M. Pedram, “Multi-dimensional SLA-Based Resource Allocation for Multi-tier Cloud Computing Systems”, *IEEE International Conference on Cloud Computing (CLOUD)*, Washington, DC, USA, 4-9 July 2011, pp. 324-331.

[34] H. N. Van, F.D. Tran, and J.-M Menaud, “Performance and Power Management for Cloud Infrastructures”, *International Conference on Cloud Computing (CLOUD)*, Miami, Florida, USA, 5-10 July 2010, pp. 329-336.

[35] V. Cardellini, E. Casalicchio, F. Lo Presti, and L. Silvestri, “SLA-aware Resource Management for Application Service Providers in the Cloud”, *International Symposium on Network Cloud Computing and Applications (NCCA)*, Toulouse, France, 21-23 Nov. 2011, pp. 20-27.

[36] L.F. Bittencourt, C.R. Senna, and E.R.M. Madeira, “Scheduling service workflows for cost optimization in hybrid clouds”, *International Conference on Network and Service Management (CNSM)*, Niagara Falls, ON, Canada, 25-29 Oct. 2010, pp. 394-397.

[37] X. Meng, C. Lizhen, W. Haiyang, and B. Yanbing, “A Multiple QoS Constrained Scheduling Strategy of Multiple Workflows for Cloud Computing”, *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Chengdu and Jiuzhai Valley, China, 10-12 Aug. 2009, pp. 629-634.

[38] S. Pandey, L. Wu, S. Guru, and R. Buyya, “A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments”, *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Perth, Australia, 20-23 April 2010, pp. 400–407.

[39] W-N. Chen and J. Zhang, “A set-based discrete PSO for cloud workflow scheduling with user-defined QoS constraints”, *International Conference on Systems, Man, and Cybernetics (SMC)*, Seoul, South Korea, 14-17 Oct. 2012, pp. 773-778.

[40] C. Szabo, and T. Kroeger, “Evolving multi-objective strategies for task allocation of scientific workflows on public clouds”, *IEEE Congress on Evolutionary Computation (CEC)*, Brisbane, Australia, 10-15 June 2012, pp.1-8.

[41] S. Abrishami, M. Naghibzadeh, and D.H.J. Epema, “Cost-Driven Scheduling of Grid Workflows Using Partial Critical Paths”, *IEEE Transactions on Parallel Distributed Systems*, vol. 23, no. 8, 2012, pp. 1400–1414.

[42] S. Abrishami, M. Naghibzadeh, and D.H.J. Epema, “Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds”, *Future Generation Computer Systems*, vol. 29, 2013, pp. 158-169.

[43] A. McGregor, D. Bennett, S. Majumdar, B. Nandy, J.O. Melendez, M. St-Hilaire, P. Lau, and J. Liu, “A Cloud-Based Platform for Supporting Research Collaboration”,

IEEE International Conference on Cloud Computing (CLOUD), New York, NY, USA, 27 June – 2 July 2015, pp. 1107-1110.

[44] J. Dittrich and J.-A. Quiane-Ruiz, "Efficient Big Data Processing in HadoopMapReduce", *In Proceedings of VLDB 2012/PVLDB*, vol. 5, no. 12, pp. 2014-2015 (Tutorial).

[45] M. Collins, "Hadoop and MapReduce: Big Data Analytics", *Gartner*, 14 Jan. 2011.

[46] N. Gift, "Solve cloud-related big data problems with MapReduce", *IBM*, 8 Nov. 2010. [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-bigdata/ [Accessed February 12, 2016].

[47] S. Baker, "The Two Flavors of Google", *Bloomberg Businessweek Magazine*, 12 Dec. 2007.

[48] T. White, "Hadoop: The Definitive Guide, 2nd Edition", *O'Reilly Media, Inc.*, Sebastopol, CA, USA, 2011.

[49] Apache, "Hadoop Wiki". [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy [Accessed: February 12, 2016].

[50] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, 2003, pp. 29–43.

[51] M. Jones and M. Nelson, "Moving ahead with Hadoop YARN", *IBM*. 2 July 2013. [Online]. Available: http://www.ibm.com/developerworks/library/bd-hadoopyarn [Accessed: February 20, 2016].

[52] The Apache Software Foundation, "Apache Hadoop YARN". [Online]. Available: http://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html [Accessed: April 5, 2016].

[53] H. Chang, M. Kodialam, R.R. Kompella, T.V. Lakshman, M. Lee, and S. Mukherjee, "Scheduling in mapreduce-like systems for fast completion time", *IEEE INFOCOM 2011*, Shanghai, China, 10-15 April 2011, pp. 3074-3082.

[54] A. S. Schulz, "Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds", *International Conference on Integer Programming and Combinatorial Optimization (IPCO)*, Vancouver, BC, Canada, 3-5 June 1996, pp. 301-315.

[55] X. Gao, Q. Chen, Y. Chen, Q. Sun, Y. Liu, and M. Li, "A Dispatching-Rule-Based Task Scheduling Policy for MapReduce with Multi-type Jobs in Heterogeneous Environments", *ChinaGrid Annual Conference*, Beijing, China, 20-23 Sept. 2012, pp. 17 -24.

[56] Z. Fadika and M. Govindaraju, "DELMA: Dynamically ELastic MapReduce Framework for CPU-Intensive Applications", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, USA, 23-26 May 2011, pp. 454-463.

[57] B. Palanisamy, A. Singh, and L. Liu, "Cost-Effective Resource Provisioning for MapReduce in a Cloud", *IEEE Transactions on in Parallel and Distributed Systems*, vol.26, no.5, 1 May 2015, pp. 1265-1279.

[58] D. Yoo and K. M. Sim, "A scheduling mechanism for multiple MapReduce jobs in a workflow application (position paper)", *Computing, Communications and Applications Conference (ComComAp)*, Hong Kong, China, 11-13 Jan. 2012, pp. 405-410.

[59] C. He, Y. Lu, and D. Swanson, "Matchmaking: A New MapReduce Scheduling Technique", *International Conference on Cloud Computing Technology and Science (CloudCom)*, Athens, Greece, 29 Nov. - 1 Dec. 2011, pp. 40-47.

[60] M. Hammoud and M.F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce", *International Conference on Cloud Computing Technology and Science (CloudCom)*, Athens, Greece, 29 Nov. - 1 Dec. 2011, pp. 570-576.

[61] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju, "MARLA: MapReduce for Heterogeneous Clusters", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, ON, Canada, 13-16 May 2012, pp. 49-56.

[62] Y. Liu, M. Li, N.K. Alham, S. Hammoud and M. Ponraj, "Load balancing in MapReduce environments for data intensive applications", *International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, Shanghai, China, 26-28 July 2011, pp. 2675-2678.

[63] X. Xu and M. Tang, "A New Approach to the Cloud-based Heterogeneous MapReduce Placement Problem", *IEEE Transactions on in Services Computing*, vol. PP, no.99, 15 May 2015, pp. 1-12.

[64] The Apache Software Foundation, "Hadoop 1.2.1 Documentation: Fair Scheduler". [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html [Accessed: April 10, 2016].

[65] The Apache Software Foundation, "Hadoop 1.2.1 Documentation: Capacity Scheduler". [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html [Accessed: April 10, 2016]

[66] Z. Guo and G. Fox, "Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization", *IEEE/ACM International*

Symposium on Cluster, Cloud and Grid Computing (CCGrid), Ottawa, ON, Canada, 13-16 May 2012, pp. 714-716.

[67] Y. Luo and B. Plale, "Hierarchical MapReduce Programming Model and Scheduling Algorithms", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid),* Ottawa, ON, Canada, 13-16 May 2012, pp. 769-774.

[68] T. Wirtz and R. Ge, "Improving MapReduce energy efficiency for computation intensive workloads", *International Conference on Green Computing Conference and Workshops (IGCC)*, Orlando, FL, USA, 25-28 July 2011, pp. 1-8.

[69] D. Cavdar, L.Y. Chen, and F. Alagoz, "Green MapReduce for heterogeneous data centers", *IEEE Global Communications Conference (GLOBECOM)*, Austin, TX, USA, 8-12 Dec. 2014, pp.1120-1126.

[70] A. Verma, L. Cherkasova, V.S. Kumar, and R.H. Campbell, "Deadline-based workload management for MapReduce environments: Pieces of the performance puzzle", *Network Operations and Management Symposium (NOMS)*, Maui, Hawaii, USA, 16-20 April 2012, pp. 900-905.

[71] K. Kc and K. Anyanwu, "Scheduling Hadoop Jobs to Meet Deadlines", *International Conference on Cloud Computing Technology and Science (CloudCom)*, Indianapolis, IN, USA, 30 Nov. – 3 Dec. 2010, pp. 388-392.

[72] X. Dong, Y. Wang, and H. Liao, "Scheduling Mixed Real-Time and Non-real-Time Applications in MapReduce Environment", *International Conference on Parallel and Distributed Systems (ICPADS)*, Tainan, Taiwan, 7-9 Dec. 2011, pp. 9-16.

[73] M. Mattess, R.N. Calheiros, and R. Buyya, "Scaling MapReduce Applications Across Hybrid Clouds to Meet Soft Deadlines", *International Conference on Advanced Information Networking and Applications (AINA)*, Barcelona, Spain, 25-28 March 2013, pp. 629-636.

[74] E. Hwang and K. H. Kim, "Minimizing Cost of Virtual Machines for Deadline-Constrained MapReduce Applications in the Cloud", *International Conference on Grid Computing (GRID)*, Beijing, China, 20-23 Sept. 2012, pp. 130-138.

[75] Z.-R. Lai, C.-W. Chang, X. Liu, T.-W. Kuo, and P.-C. Hsiu, "Deadline-aware load balancing for MapReduce", *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Chongqing, China, 20-22 Aug. 2014, pp. 1-10.

[76] C. Chen, J. Lin, and S. Kuo, "MapReduce Scheduling for Deadline-Constrained Jobs in Heterogeneous Cloud Computing Systems", *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1-14.

[77] U. Farooq, S. Majumdar, and E. W. Parsons, “Achieving efficiency, quality of service and robustness in multi-organizational Grids”, *Journal of Systems and Software*, vol. 82, Jan. 2009, pp. 23–38.

[78] W. Tang, N. Desai, D. Buettner, and Z. Lan, “Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P”, *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, USA, 19-23 April 2010, pp. 1–11.

[79] P. Xiao and Z. Hu, “Relaxed resource advance reservation policy in grid computing”, *The Journal of China Universities of Posts and Telecommunications*, vol. 16, no. 2, April 2009, pp. 108–113.

[80] G. Birkenheuer, A. Brinkmann, and H. Karl, “Risk aware overbooking for commercial grids”, *International workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, Atlanta, GA, USA, 23 April 2010, pp. 51–76.

[81] P. Hoang, S. Majumdar, M. Zaman, P. Srivastava, and N. Goel, “Resource Management Techniques for Handling Uncertainties in User Estimated Job Execution Times”, *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Monterey, CA, USA, 6-10 July 2014, pp. 626 -633.

[82] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “Backfilling Using System-Generated Predictions Rather than User Runtime Estimates”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, 2007, pp. 789–803.

[83] A. Matsunaga and J. Fortes, “On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications”, *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, Melbourne, Australia, 17-20 May 2010, pp. 495 –504.

[84] Y. Murata, R. Egawa, M. Higashida, and H. Kobayashi, “A History-Based Job Scheduling Mechanism for the Vector Computing Cloud”, *IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*, Seoul, South Korea, 19-23 July 2010, pp. 125 –128.

[85] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, “Meeting service level objectives of Pig programs”, *International Workshop on Cloud Computing Platforms*, New York, NY, USA, 2012, pp. 8:1–8:6.

[86] B. Li, J. Chen, M. Yang, and E. Wang, “Impact of Extending the Runtime of Underestimated Jobs in Backfilling Schedulers”, *International Conference on Computer Science and Software Engineering (CSSE)*, Wuhan, China, 12-14 Dec. 2008, pp. 328–331.

[87] J. Polo, “Adaptive Scheduler”, [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-1380 [Accessed: February 24, 2016].

[88] J.N. Hooker, “Planning and scheduling to minimize tardiness”, P. In van Beek, ed., *Principles and Practice of Constraint Programming*, vol. 3709, 2005, pp. 314–327.

[89] I. J. Lustig and J.-F. Puget, “Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming”, *INTERFACES*, vol. 31, no. 6, Nov.-Dec. 2001, pp. 29-53.

[90] P. Refalo, “Linear formulation of constraint programming models and hybrid solvers”, *Principles and Practice of Constraint Programming–CP 2000*, Springer Berlin Heidelberg, 2000, pp. 369-383.

[91] N. Beldiceanu and S. Demassey, “Global Constraint Catalog”. [Online]. Available: http://sofdem.github.io/gccat/gccat/Ccumulative.html [Accessed: February 25, 2016].

[92] Lindo Systems Inc., “Lindo Systems – Optimization Software”. [Online]. Available: http://www.lindo.com/ [Accessed: February 25, 2016].

[93] NICTA, “MiniZinc and FlatZinc”. [Online]. Available: http://www.MiniZinc.org/ [Accessed: February 24, 2016].

[94] Gecode, “Generic Constraint Development Environment”. [Online]. Available: http://www.gecode.org/ [Accessed: February 25, 2016].

[95] J.M. Van den Akker, C. Hurkens, and M. Savelsbergh, “Time-indexed formulations for machine scheduling problems: Column generation”, *INFORMS Journal on Computing*, vol. 12, no. 2, 2000, pp. 111-124.

[96] K. Marriott, P.J. Stuckey, L.D. Koninck, and H. Samulowitz, “An Introduction to MiniZinc Version 1.6”. [Online]. Available: www.minizinc.org/downloads/doc-1.6/minizinc-tute.pdf [Accessed February 25, 2016].

[97] IBM Corporation, “IBM ILOG CPLEX Optimization Studio V12.5 Reference Manual”. [Online]. Available: http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r5/index.jsp [Accessed: February 25, 2016].

[98] IBM Corporation, “Detailed Scheduling in IBM ILOG CPLEX Optimization Studio with IBM ILOG CPLEX CP Optimizer”, *White Paper*. IBM Corporation, 2010. [Online]. Available: http://www.besmart.company/wp-content/uploads/2014/11/Ilog-CPLEX.pdf [Accessed: February 25, 2016].

[99] T. Dong, “Efficient modeling with the IBM ILOG OPL-CPLEX Development Bundles”, *White Paper*, December 2009. [Online]. Available: http://public.dhe.ibm.com/common/ssi/rep_wh/n/WSW14059USEN/WSW14059USEN.PDF [Accessed: February 25, 2016].

[100] Oracle Corporation, “NetBeans IDE”, [Online]. Available: https://netbeans.org [Accessed: February 25, 2016].

[101] R. Jain, “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling”, Wiley-Interscience, New York, NY, April 1991, ISBN:0471503361.

[102] Oracle Corporation, “System.nanoTime()”, *Java Platform Standard Ed. 7*. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime() [Accessed: February 24, 2016].

[103] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling”, *EuroSys*, Paris, France, 13-16 April 2010, pp. 265–278.

[104] M. Jones, “Scheduling in Hadoop”, [Online]. Available: http://www.ibm.com/developerworks/library/os-hadoop-scheduling/ [Accessed: February 25, 2016].

[105] Oracle Corporation, “Interface Comparator<T>”, *Java Platform Standard Ed. 7*. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html [Accessed: February 25, 2016].

[106] Oracle Corporation, “Nested Classes”, *The Java Tutorials*. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html [Accessed: February 25, 2016].

[107] The IEEE and the Open Group, “Seconds Since the Epoch”, *The Open Group Base Specifications Issue 7*, IEEE Std 1003.1, 2013.

[108] Amazon, “Amazon EC2 Instance Types”. [Online]. Available: http://aws.amazon.com/ec2/instance-types/ [Accessed: February 25, 2016].

[109] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis”, *IEEE International Conference on Data Engineering Workshops (ICDEW)*, Long Beach, CA, USA, 1-6 March 2010, pp. 41-51.

[110] Oracle Corporation, “Thread.sleep()”, *Java Platform Standard Ed. 7*. [Online]. Available:

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#sleep(long) [Accessed: October 15, 2016].

[111] D. England, J. Weissman, and J. Sadagopan, "A new metric for robustness with application to job scheduling", *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Research Triangle Park, NC, USA, 24-27 July 2005, pp. 135 – 143.

[112] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling", *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, Jun. 2001, pp. 529 –543.

[113] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of Scientific Workflows", *Workshop on Workflows in Support of Large Scale Science*, Austin, TX, USA, 17 Nov 2008, pp. 1-10.

[114] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and Profiling Scientific Workflows", *Future Generation Computer Systems*, vol. 29, no. 3, March 2013, pp. 682–692.

[115] K. J. Lin, S. Natarajan, and J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems", *IEEE Real-Time Systems Symposium (RTSS)*, 1-3 Dec. 1987, San Jose, CA, USA, pp. 210-217.

[116] G. L. Stavrinides and H. D. Karatza, "Scheduling real-time parallel applications in SaaS clouds in the presence of transient software failures", *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Montreal, QC, Canada, 24-27 July 2016, pp. 1-8.

[117] Oracle Corporation, "Interface Iterator<E>", *Java Platform Standard Ed. 7*. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html [Accessed: February 26, 2016].

Appendix A Design and Implementation of the MILP Model and the CP Model

The following sub-sections (A.I to A.III) describe the design and implementation of the MILP Model and the CP Model using various commercial-off-the-shelf and open source software packages.

A.I. MILP Model Implemented Using LINGO

This section discusses how the MILP Model is implemented in LINGO v13.0. Additional information on how to use LINGO is found in [92]. The LINGO modeling language provides a data type called *Sets* that is used to model a group of related objects. By using Sets, constraints on the decision variables are efficiently and compactly expressed using a single statement. Each set can have one or more attributes associated with each item of the set. In the implementation of the MILP Model, sets are used to represent the input: jobs set J, tasks set AT, resources set R, and time range set I. For example, the task set AT is implemented as follows:

```
SETS: TASKS: parentJob, type, execTime, resReq;
```

The *parent job* attribute identifies which job the task belongs to. For example, if the parent job attribute of a task is 2, it means that this task belongs to the job with an id equal to 2. The *type* attribute indicates whether the task is a map task (type = 0) or a reduce task (type = 1). The `execTime` and `resReq` attributes represent e_t and q_t, respectively.

A representative set of examples of how the constraints of the MILP Model (defined in Table 3.2) are implemented using LINGO are presented. For instance, constraint (1b) is implemented as follows:

```
@FOR( TASKS(t):
      @SUM( TIME(i):
            @SUM( RESOURCES(r): x(t,r,i) )
      ) = 1
);
```

The @FOR construct is used to iterate the members of a given set and can be used to generate constraints for each member of the set. As the name suggests, the @SUM construct is a looping function that calculates the sum of all the members in the given set. The variable *x* used in the LINGO model represents the *x* decision variable discussed in Section 3.4.

The implementation of constraint (5b) using LINGO is presented:

```
@FOR( RESOURCES(r):
    @FOR(TIME(i):
         @SUM( TASKS(t)| type(t) #EQ# 0:
              @SUM( TIME(i2 |(i-execTime(t)) #LT# i2 #AND# i2 #LE# i:
                   x(t,r,i2)*resReq(t)
              )
         ) <= mapCapacity(r)
    )
);
```

As shown, the two @SUM constructs use LINGO's conditional qualifier operator |, which limits the scope of the looping function and restricts the members of the set that are processed. More specifically, only the members of the set that evaluate the conditional qualifier equation to true are processed. For example, the first @SUM construct specifies that only tasks with a type attribute equal to 0 (i.e., map tasks) are processed. Furthermore, the syntax for using logical operators is to enclose the name of the operator using hashtags (#). For instance, the syntax for the logical operators: =, ≤, and < are #EQ#, #LE#, and #LT#, respectively.

A.II. CP Model Implemented Using MiniZinc

This section discusses how the CP Model is implemented using MiniZinc. More details on how to use the MiniZinc modeling language can be found in [96]. Similar to

LINGO, MiniZinc also provides a mechanism to group together closely related data called *Sets and Arrays.* Using MiniZinc, the set of tasks *AT* is implemented as follows:

```
set of int: Tasks = 1..NUM_TASKS;
set of int: Jobs = 1..NUM_JOBS;
array [Tasks] of Jobs: parentJob;
array [Tasks] of 0..1: type;
array [Tasks] of int: execTime;
array [Tasks] of int: resourceReq;
```

First a set of integers, called `Tasks`, is defined from 1 to a variable named `NUM_TASKS`, which stores the total number of tasks contained in the input set of jobs, *J*. Next, the attributes of the tasks, which are identical to the attributes described in Appendix A.I, are declared using arrays. The indices of the attribute arrays are specified using the set of integers, `Tasks`, that was declared previously. For example, the `execTime` array has its indices defined by the `Tasks` set so that each task has its own `execTime` attribute. The domain of each of the attributes, which is the range of acceptable values that an attribute can have, is also defined when declaring the attribute arrays by using the `of` keyword. For example, the domain of the `parentJob` attribute is equal to the set of integers called `Jobs`, which has a range from 1 to `NUM_JOBS` where `NUM_JOBS` is a variable that stores the number of jobs in the input set *J*. It is observed from the above example that the implementation of data sets in MiniZinc requires using two data types (sets and arrays), and is not as compact as the one used in LINGO, but it performs the same function.

A representative set of examples of how the CP Model's constraints (defined in Table 3.1) are implemented using MiniZinc is provided next. In MiniZinc, constraint (2a) is expressed as follows:

```
constraint forall(j in Jobs) (
    forall(t in Tasks where parentJob[t] == j /\ type[t] == 0) (
        startTime[t] >= releaseTime[j]
    )
);
```

All the constraints in MiniZinc, start with the keyword `constraint`. The `forall` construct performs an identical function to LINGO's `@FOR` construct, which is used to iterate the items of a collection. Similarly, the `where` keyword in the second `forall` statement is MiniZinc's conditional qualifier operator. The `/\` operator performs a logical conjunction (logical `and`) operation.

A.III. CP Model Implemented Using IBM CPLEX

In this section, the implementation of the CP Model using OPL, which is referred to as the *OPL Model*, is discussed. Additional information for expressing CP formulations using OPL can be found in [97] and [98]. Similar to the other approaches, OPL supports using a data type called `tuple`, which allows related data to be grouped together. The following tuples are defined to represent a job, a task, a resource, and the x_{tr} decision variable, respectively:

- `Job` = <`id`, `earliestStartTime`, `deadline`>,
- `Task` = <`id`, `parentJob`, `type`, `execTime`, `resReq`>,
- `Resource` = <`id`, `mapCapacity`, `reduceCapacity`>, and
- `Option` = <`Task`, `Resource`>.

For example, the `Tasks` set is expressed in OPL as follows:

```
tuple Task {
   key string id;
   int parentJob;
   int type;
   int execTime;
   int resReq;
};

{ Task } Tasks = ...;
```

First, a `Task` tuple is defined, which is then used to specify a set of `Task` tuples called `Tasks`. The ellipses (…) indicate that the `Tasks` set will be specified as input to the model. The `Task` tuple has the same attributes as those discussed in Appendix A.I, except for an additional attribute called `id`, which is used to uniquely identify a tuple in OPL. The keyword `key` specifies which attribute is used as the unique identifier for the tuple.

A representative set of examples of implementing the constraints of the CP Model (recall Table 3.1) is presented. In the OPL Model, constraint (1a) is expressed using the OPL-defined `alternative` constraint [98] as follows:

```
forall (t in Tasks)
    alternative(taskIntervals[t],
        all(o in Options: o.task.id == t.id) xtr[o]);
```

Similar to the `forall` construct used in MiniZinc, OPL's `forall` construct is used to iterate through the elements of a specified collection. The `alternative` constraint is a synchronization constraint that requires two parameters: an `interval i` and a set of intervals `S`. The `alternative` constraint states that the interval `i` will only be present in the solution if and only if there is exactly one interval in *S* (denoted `j`) that is also present in the solution. Both intervals `i` and `j` are synchronized meaning that they both start and end at the same time. Thus, it is appropriate to use the `alternative` constraint to express constraint (1a) whose purpose is to ensure that each task is assigned to only one resource (recall Section 3.3). In the above example, the set `S` is produced by using the `all` construct invoked with a conditional qualifier (`:`). More specifically, `S` is a subset of `Option` tuples that have the same id as the task of interest, *t*.

In the OPL Model, constraint (5a) is expressed as follows:

```
forall (r in Resources) {
      sum (o in Options : o.resource.id == r.id && o.task.type == 0)
            pulse(xtr[o], o.task.resReq) <= r.mapCapacity;
}
```

The `pulse` function [98] is used to generate the resource usage of a task, and it requires two parameters: an interval `i` that represents the task and a height value `h` to indicate the resource usage (i.e., capacity requirement) of the task. The pulse function generates a value as a function of time. More precisely, when the task is running (i.e., during the time between the interval's start time and end time), the pulse function generates a value equal to the supplied value `h` to indicate the amount of resource usage of the task. At all other points in time, the pulse function generates a value of 0. To implement the functionality of CP's `cumulative` constraint, the OPL Model sums all the values produced by the `pulse` function at each point in time and asserts that the sum is less than or equal to the capacity of the resource. As shown, the OPL implementation of constraint (5a), which enforces that the map task capacity of each resource *r* in *R* is not violated, uses a conditional qualifier (:) to ensure that only the map tasks (represented by the `xtr[o] interval` variable) that are assigned to the resource of interest (resource *r*) is included in the calculation. To determine that if an `xtr[o]` variable represents a map task that is scheduled on resource *r*, `xtr[o]`'s `task.type` and `resource.id` attributes are checked to see if they are equal to 0 and are equal to resource *r*'s id, respectively.

Appendix B Additional Details on the MRCP-RM Algorithm

In this appendix, additional details of the design and implementation of the MRCP RM algorithm are provided.

B.I. Creating and Solving the OPL Model Using IBM CPLEX's Java APIs

This section describes how IBM CPLEX's Java APIs are used to create and solve an OPL Model, including how the solution of the OPL Model is extracted and saved. The CPLEX Java library follows a Factory design pattern [97] where objects are created by invoking methods from a single master object. Thus, the first step is to create an instance of an `IloOplFactory` object as follows:

```
IloOplFactory factory = new IloOplFactory();
```

Next, the model source object, `IloOplModelSource`, is created using:

```
IloOplModelSource modelSource =
     factory.createOplModelSourceFromString(oplModelText, modelName);
```

where `oplModelText` is a string containing the source code (text) of the OPL Model and `modelName` is a string containing the user-specified name of the model. The model definition object can then be created:

```
IloOplModelDefinition modelDef =
     factory.createOplModelDefinition(modelSource, settings)
```

where `modelSource` is the `IloOplModelSource` object that was previously created and `settings` is an instance of `IloOplSettings` that represents the configuration and settings of the model. The `IloOplSettings` object is created as follows:

```
IloOplErrorHandler errHandler = factory.createOplErrorHandler();
IloOplSettings settings = factory.createOplSettings(errHandler);
```

As shown, an error handler object, `IloOplErrorHandler`, needs to be created before generating the `settings` object. The next step is to create an instance of the CP Optimizer solving engine:

```
IloCP cpSolver = factory.createCP()
```

The OPL Model object can then be created by invoking the `createOplModel()` method, which requires passing in a model definition object and a CP Optimizer instance as follows:

```
IloOplModel oplModel = factory.createOplModel(modelDef, cpSolver)
```

Now that the OPL Model object is created, a data source that provides the input data for the model is added using:

```
IloOplDataSource dataSource = new OPLModelData(factory, resources,
                                                jobsToSchedule);
oplModel.addDataSource(dataSource);
```

`OPLModelData` is a user-defined class that extends CPLEX's `ilog.opl.IloCustomOplDataSource` class [97] whose purpose is to generate the OPL Model's input data: set of jobs to schedule J and set of resources R. The `IloOplModel` object can now be generated as follows:

```
oplModel.generate()
```

This converts the model into a form that the CP Optimizer can solve. Finally, the OPL Model is solved by invoking:

```
cpSolver.solve()
```

After a solution is found, the values of the decision variables (i.e., the assigned resource and scheduled start time of each scheduled task) are extracted from the solved OPL Model using CPLEX's Java API OPL element interface [97]. A discussion on how this is done is provided next. First, the element that contains all the scheduled tasks is obtained from the `IloOplModel` object using:

```
IloTupleSet scheduledTasks =
      oplModel.getElement("ScheduledTasks").asTupleSet();
```

Next the `scheduledTasks` set, which is an instance of `IloTupleSet` and contains a set of `ScheduledTask` tuples, is processed to retrieve the values of the decision variables using:

```
for (Iterator it = scheduledTasks.iterator(); it.hasNext(); )
{
      IloTuple tuple = (IloTuple) it.next();

      String taskId = tuple.getStringValue("taskID");
      int assignedResId = tuple.getIntValue("resID");
      int startTime = tuple.getIntValue("start");
      int endTime = tuple.getIntValue("end");

      ...
}
```

The `ScheduledTask` tuple is used by the post-processing component of the OPL Model to store the values of the decision variables and is defined as follows:

```
tuple  ScheduledTask
{
  int resID;
  int isReduceTask;
  int start;
  int end;
  string taskID;
};
```

The post-processing component of the OPL Model is executed after the CP Optimizer finds a solution to the OPL Model, and its job is to save the values of the decision variables into the attributes of the `ScheduledTask` tuple. As shown, to iterate the `scheduledTasks` set, Java's `Iterator` class [117] is used. The values of the `ScheduledTask` tuples are retrieved by using the `IloTuple` class' `get` methods which include `getIntValue()` to retrieve an integer and `getStringValue()` to retrieve a string.

B.II. Split Single Resource Schedule Algorithm

This section describes the Split Single Resource Schedule algorithm (see Algorithm B.1) that is used by the MRCP-RM technique (refer to Section 4.3.1). The input required by the algorithm includes the following: the single combined resource (*sr*), *nmRes*, the number of *map resources*, which are resources with a map task capacity, $c_r^{mp} \geq 1$ in *R*, and *nrRes*, the number of *reduce resources*, which are resources that have a reduce task capacity, $c_r^{rd} \geq 1$), in *R*.

The first phase of Algorithm B.1 (lines 1-5) moves the map tasks and reduce tasks from the single combined resource to a *set of single capacity map resources* (*MR*) and a *set of single capacity reduce resources* (*RR*), respectively. Each resource in *MR* has only one map task slot and each resource in *RR* has only one reduce task slot. The number of single capacity resources in *MR* and *RR* is equal to the total number of map task slots in *R* ($totMC = \sum_{r \in R} c_r^{mp}$) and the total number of reduce task slots in *R* ($totRC = \sum_{r \in R} c_r^{rd}$), respectively. The algorithm moves tasks from the single combined resource in non-decreasing order of the respective scheduled start times of the tasks and assigns each task to a single capacity resource in *MR* or *RR*, depending on whether it is a map task or a reduce task. More specifically, each task *t* is assigned to the single capacity resource that is found in *MR* or *RR* that leaves the smallest remaining *gap* in the resource's schedule. For example, consider a scenario in which there are two resources: *r1* and *r2*, and a task $\tau_{3,1}$ that needs to be assigned to one of these two resources from time 10 to 15. Resource *r1* already has a task $\tau_{1,1}$ scheduled from time 2 to 10 and *r2* already has a task $\tau_{2,1}$scheduled from time 5 to 8. Task $\tau_{3,1}$would be assigned to *r1* since the resulting gap will be 10 – 10 = 0 compared to the gap for *r2*: 10 – 8 = 2.

Algorithm B.1: MRCP-RM algorithm's *splitSingleResourceSchedule()*

Input: single combined resource (*sr*), integer *nmRes*, and integer *nrRes*
Output: a list of resources

```
1:  totMC ← sr.getMapCapacity()
2:  totRC ← sr.getReduceCapacity()
3:  Create a set of single capacity map resources, MR, comprising totMC single
    capacity map resources.
4:  Create a set of single capacity reduce resources, RR, comprising totRC single
    capacity reduce resources.
5:  Assign map tasks and reduce tasks from sr to MR and RR, respectively.
6:  minMC ← totMC / nmRes ;  leftOverMC ← totMC – minMC * nmRes
7:  minRC ← totRC / nrRes ;  leftOverRC ← totRC – minRC * nrRes
8:  id ← 1
9:  Create newResources list.
10: totMC ← totMC - leftOverMC
11: totRC ← totRC – leftOverRC
12: while (totMC > 0 || totRC > 0) do
13:     mc ← (totMC > 0 ? minMC : 0) + (leftOverMC > 0 ? 1 : 0)
14:     rc ← (totRC> 0 ? minRC : 0) + (leftOverRC > 0 ? 1 : 0)
15:     res ← new Resource(id, mc, rc)
16:     newResources.add(res)
17:     totMC ← totMC  - minMC ; totRC← totRC - minRC
18:     leftOverMC ← leftOverMC – 1 ; leftOverRC ← leftOverRC - 1
19:     id ← id + 1
20: end while
21: for each resource r in newResources do
22:     Assign scheduled map tasks from num single capacity resources in MR, where
        num is equal to r.getMapCapacity().
23:     Assign scheduled reduce tasks from num single capacity resources in RR,
        where num is equal r.getReduceCapacity().
24: end for
25: return newResources
```

The second phase of Algorithm B.1 (lines 6-7) calculates the number of map task slots and the number of reduce task slots for each new resource that will be created to represent the original resources in *R*. Recall that the input to the algorithm includes the number of resources with at least one map task slot (*nmRes*) and the number of resources with at least one reduce task slot (*nrRes*). The total number of map task slots and the total number of reduce task slots is divided evenly among the resources. In line 6, the minimum

number of map task slots for each resource (stored in the variable *minMC*) and the left over total map task capacity (stored in the variable *leftOverMC*), which is the remaining number of map task slots to be assigned, are calculated. Similarly, in line 7 the minimum number of reduce task slots for each resource (stored in a variable *minRC*) and the left over total reduce task capacity (stored in a variable *leftOverRC*), which is the remaining number of reduce task slots to be assigned, are calculated.

In the third phase of Algorithm B.1 (lines 8-20), the new resources are created. First, a local variable named *id* is initialized to 1 and a new list of resources, named *newResources*, is created (lines 8-9). The *totMC* and *totRC* variables are then updated by subtracting *leftOverMC* and *leftOverRC* variables, respectively, as shown in lines 10-11. The while loop shown in lines 12-20 is used to create the new resources for the system and the loop continues as long as the variables *totMC* or *totRC* are greater than 0 (i.e., there is still more resources to create). The map task capacity and reduce task capacity for each resource (stored in variables *mc* and *rc*, respectively) are then calculated as shown in lines 13-14. If *totMC* is greater than 0, the new resource will have a map task capacity equal to *minMC*; otherwise, the map task capacity is equal to 0. Furthermore, if *leftOverMC* is greater than 0, the resource will get an additional map task slot. Similar calculations are then performed to calculate the reduce task capacity of the resource (see line 14). The next step is to create a new resource with an id equal to *id*, a map capacity equal to *mc*, and a reduce capacity equal to *rc* (line 15). This new resource is then added to the *newResources* list (line 16). The last step of the while loop is to update variables *totMC*, *totRC*, *leftOverMC*, *leftOverRC*, and *id* as shown in lines 17-19.

The last phase of Algorithm B.1 (lines 21-25) assigns tasks to each of the new resources created. More specifically, each resource *r* in *newResources* is assigned scheduled map tasks and scheduled reduce tasks from the single capacity resources in *MR* and the single capacity resources in *RR*: the numbers of which are equal to resource *r*'s map task capacity and reduce task capacity, respectively. The *newResources* list is then returned and the algorithm ends (line 25).

Appendix C Additional Details on the Design and Implementation of the Hadoop CP-Scheduler

The focus of this appendix is on providing additional details of the modifications made to Hadoop 1.2.1 to implement the Hadoop CP-Scheduler (described in Section 5.3). This includes a discussion on adding support for job deadlines in Hadoop (see Appendix C.I) and a discussion on adding support for user-estimated task execution times (see Appendix C.II). In addition, details on the following are provided: (1) integrating IBM CPLEX with Hadoop (see Appendix C.III) and (2) a discussion of the `createNewModelDefinition()` method that is used by the HCP-RM algorithm (see Appendix C.IV).

C.I. Adding Support for Job Deadlines

This section discusses the Hadoop classes that are modified to support user-specified job deadlines. First, in Hadoop's `org.apache.hadoop.mapred.JobInProgress` class, a new private field, `long deadline`, is added to store a job's deadline. The value stored in the deadline field represents the number of milliseconds elapsed from midnight, January 1, 1970 UTC (known as the Unix Epoch [107]). Recall from Section 5.3.3 that the `JobInProgress` class represents a MapReduce job that is being tracked by JobTracker. The `deadline` field of the `JobInProgress` class is initialized via its constructor by invoking `conf.getJobDeadline()` where `conf` is an object that is an instance of the `org.apache.hadoop.mapred.JobConf` class and `getJobDeadline()` is a new method implemented in the `JobConf` class whose purpose is to retrieve the job's deadline.

The `JobConf` class represents a MapReduce job configuration file. It is an interface for users to specify the properties (e.g., job name and the number of map and reduce tasks) for their MapReduce job before submission to the Hadoop cluster. Two new methods are added to the `JobConf` class: `getJobDeadline()` and `setJobDeadline(long deadline)`. The method `setJobDeadline(long deadline)` sets a new job configuration property, `mapred.job.deadline`, to the supplied parameter. On the other hand, the `getJobDeadline()` method is used to retrieve the value assigned to the `mapred.job.deadline` property.

The other Hadoop class that is modified to support user-specified job deadlines is the `org.apache.hadoop.mapreduce.Job` class. The `Job` class provides a user API, and it is the class that a user uses to create and submit a job to the Hadoop cluster. The `Job` class also supplies methods to allow the user to configure the job, control its execution, and obtain status information (e.g., state of the job). Similar to the `JobConf` class, the two new methods added to the `Job` class are: `setJobDeadline(long deadline)` and `getJobDeadline()`. These two methods in turn invoke `conf.setJobDeadline(deadline)` and `conf.getJobDeadline()`, respectively, where `conf` is an instance of a `JobConf` object. Note that `conf` is one of the private fields of the `Job` class and is initialized when a `Job` object is created. In summary, the sequence of calls for setting the deadline of a job is illustrated in the sequence diagram shown in Figure C.1.

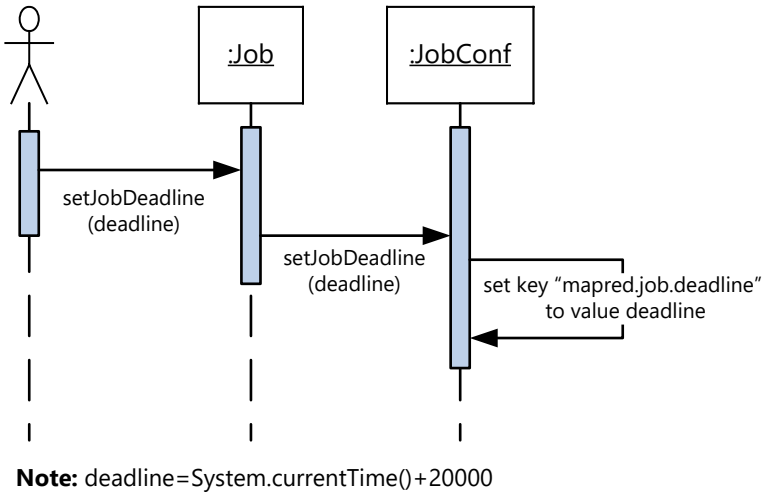

Figure C.1. Sequence diagram for setting the deadline of a job in Hadoop.

C.II. Adding Support for User-estimated Task Execution Times

Similar to how support for job deadlines is added to Hadoop, support to allow users to specify the estimated task execution times of their submitted jobs is accomplished by adding two new methods: `setEstimatedTaskExecutionTimes(String execTimes, int taskType)` and `getEstimatedTaskExecutionTimes(int taskType)` (abbreviated `setET` and `getET`, respectively) to Hadoop's `Job` and `JobConf` classes.

The `setET` method accepts two parameters: a comma delimitated string of task execution times in seconds (e.g., "2,2,3"), and the task type (map = 0 or reduce = 1). Depending on the task type, the `setET` method assigns either the `mapred.job.mapTaskExecTimes` property or the `mapred.job.reduceTaskExecTimes` property to the supplied string, `execTimes`. The `getET` method accepts a single parameter, the task type (map = 0 or reduce = 1), and it returns a string array containing the values assigned to the corresponding property. In summary, the sequence of calls for retrieving

the estimated task execution times of a job is illustrated in the sequence diagram shown in Figure C.2.

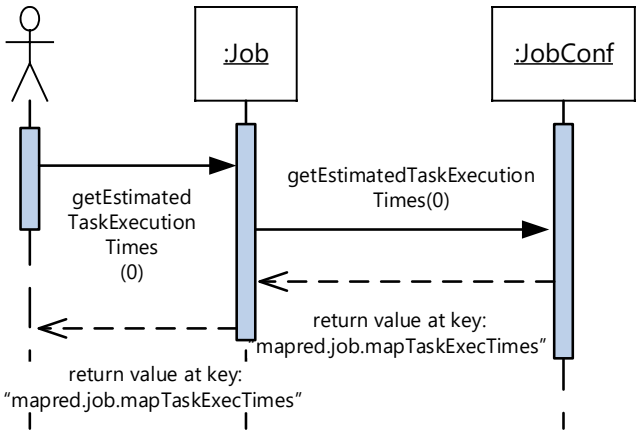

Figure C.2. Sequence diagram for retrieving the estimated task execution times of a job in Hadoop.

C.III. Details on Integrating IBM CPLEX with Hadoop

This section describes how IBM CPLEX is integrated with Hadoop. To import the required CPLEX Java libraries, IBM CPLEX v12.5 is first installed on the Ubuntu (Linux) machine where the CP-Scheduler is executed on. Next, the IBM CPLEX v12.5 JAR (Java archive) file, named `oplall12.5.jar`, is placed in Hadoop's `/hadoop/lib` folder. In addition, a modification is made to Hadoop's `/hadoop/bin/hadoop` script so that the JobTracker can locate the CPLEX libraries. More specifically, the `java.library.path` variable of the `hadoop` script is modified to include the folder:

```
<IBM_CPEX_Install_dir>/opl/bin/x86-64_sles10_4.1
```

C.IV. Create New Model Definition Method

The `createNewModelDefinition()` method, which is used by the HCP-RM algorithm (see Section 5.4.2) is presented in Algorithm C.1. The first step is to initialize the variable `modelSrc` with a string containing the OPL Model's source code (line 1), which

is obtained from the `OPLModelSource` class (recall Section 5.3.2). The next step is to process all the scheduled tasks in the system (represented by `Task_CPS` objects) to check the state of each of the tasks' corresponding `TaskInProgress` object (abbreviated `TIP`) (lines 2 to 12). If a task *t*'s `TIP` is in the *running* state, the CP-Scheduler's `addConstraints()` method is invoked (line 6). This method adds a new constraint to `modelSrc` to specify that the time interval from task *t*'s scheduled start time to its scheduled completion time on its assigned resource *r* is occupied. The purpose of the new constraint is to prevent the CP Optimizer solving engine from scheduling other tasks on resource *r* during *t*'s scheduled time, if the resource does not have the capacity to process additional tasks. In addition, the `Task_CPS`' `isExecuting` field is set to true (line 7), which is passed on to the OPL Model (via `OPLModelData` class) to tell the CP Optimizer that enforcing constraint (2a) is not required for tasks that are already executing (recall the discussion in Section 4.1.1).

Algorithm C.1: CP-Scheduler's *createNewModelDefinition()*

Input: none
Output: none

```
1:  modelSrc ← OPLModelSource.getSource()
2:  for each resource r in resources do
3:      for each task t in r.getAllScheduledTasks() do
4:          tip ← t.getTaskInProgress()
5:          if tip is in the running state then
6:              call addConstraints(modelSrc, t, r)
7:              t.setCurrentlyExecuting(true)
8:          else if tip is in the completed state then
9:              call removeTask(t)
10:         end if
11:     end for
12: end for
13: modelDefinition ← Create new OPL model definition using the updated
                      OPL model source, modelSrc.
```

On the other hand, if the task's `TIP` is in the *completed* state (line 8) then the CP-Scheduler's `removeTask()` method is invoked to remove the completed task from the system (line 9). After all the scheduled tasks in the system are processed, the final step of the algorithm is to create the new OPL model definition object using the updated `modelSrc` and store it in the `CP_Scheduler` class' `modelDefinition` field (line 13).

Appendix D Additional Results for the Performance Evaluation of the WFBB-RM Technique

In Appendix D.I to Appendix D.III, the complete set of results of the factor-at-a-time experiments (described in Section 7.5) are presented. This includes the results of all 8 WFBB-RM configurations for each of the three workloads (CyberShake, LIGO, and Genome). Note that for the sake of completeness, the results described in Section 7.5 are also included in the tables presented in this appendix. Furthermore, in Appendix D.IV, additional results for the experiments conducted to compare the performance of WFBB-RM with that of MRCP-RM are presented.

D.I. CyberShake Workload

The results of the factor-at-a-time experiments conducted using the CyberShake workload are presented in this section. More specifically, the results of the experiments using the PD-based WFBB-RM configurations and the ED-based WFBB-RM configurations are shown in Appendix D.I.I and Appendix D.I.II, respectively.

D.I.I *Results of Experiments Using the PD-based WFBB-RM Configurations*

Table D.1, Table D.2, Table D.3, and Table D.4 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the PD-based WFBB-RM configurations.

Table D.1. CyberShake workload: effect of λ_{CS} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

λ_{CS} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
1/30	0.002 ±0.001	0.002 ±0.001	0.574 ±0.034	0.547 ±0.034	248.27 ±0.48	248.27 ±0.48	580.23 ±1.14	580.30 ±1.09	0.010 ±0.000	0.010 ±0.000	0.016 ±0.000	0.018 ±0.000
1/22	0.195 ±0.042	0.195 ±0.042	1.075 ±0.061	1.211 ±0.064	293.99 ±2.25	293.99 ±2.25	566.12 ±0.95	568.68 ±0.95	0.016 ±0.001	0.016 ±0.001	0.019 ±0.000	0.022 ±0.000
1/18	1.824 ±0.127	1.824 ±0.127	2.487 ±0.121	2.832 ±0.135	404.99 ±9.00	404.99 ±9.00	575.80 ±2.51	581.43 ±2.72	0.045 ±0.004	0.045 ±0.004	0.025 ±0.001	0.030 ±0.002

Table D.2. CyberShake workload: effect of em_{max} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

em_{max}	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
2	4.173 ±0.112	4.173 ±0.112	7.520 ±0.140	7.899 ±0.148	266.23 ±1.14	266.23 ±1.14	321.72 ±0.60	323.13 ±0.65	0.029 ±0.001	0.029 ±0.001	0.029 ±0.000	0.031 ±0.001
5	0.207 ±0.060	0.207 ±0.060	1.075 ±0.061	1.211 ±0.064	294.52 ±3.03	294.52 ±3.03	566.12 ±0.95	568.68 ±0.95	0.016 ±0.001	0.016 ±0.001	0.019 ±0.000	0.022 ±0.000
10	0.000 ±0.001	0.000 ±0.001	0.107 ±0.014	0.115 ±0.017	301.60 ±1.71	301.60 ±1.71	941.27 ±1.88	947.05 ±1.91	0.012 ±0.000	0.012 ±0.000	0.020 ±0.000	0.024 ±0.000

Table D.3. CyberShake workload: effect of s_{max} on P, T, and O when using the PD-based WFBB-RM configurations.

s_{max} (sec)	P (%)				T (sec)				O (sec)			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
10000	0.514 ±0.062	0.514 ±0.062	1.948 ±0.089	2.250 ±0.100	357.15 ±2.72	357.15 ±2.72	547.67 ±0.96	552.78 ±0.96	0.022 ±0.001	0.022 ±0.001	0.024 ±0.000	0.027 ±0.000
50000	0.207 ±0.060	0.207 ±0.060	1.075 ±0.061	1.211 ±0.064	294.52 ±3.03	294.52 ±3.03	566.12 ±0.95	568.68 ±0.95	0.016 ±0.001	0.016 ±0.001	0.019 ±0.000	0.022 ±0.000
250000	0.000 ±0.000	0.000 ±0.000	0.328 ±0.024	0.308 ±0.023	231.85 ±0.23	231.85 ±0.23	619.55 ±1.02	619.11 ±1.01	0.008 ±0.000	0.008 ±0.000	0.013 ±0.000	0.015 ±0.000

Table D.4. CyberShake workload: effect of m on P, T, and O when using the PD-based WFBB-RM configurations.

m	P (%)				T (sec)				O (sec)			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
40	4.367 ±0.162	4.367 ±0.162	5.122 ±0.155	5.633 ±0.156	706.42 ±23.82	706.42 ±23.82	614.08 ±7.46	625.32 ±8.24	0.088 ±0.008	0.088 ±0.008	0.029 ±0.002	0.034 ±0.003
50	0.207 ±0.060	0.207 ±0.060	1.075 ±0.061	1.211 ±0.064	294.52 ±3.03	294.52 ±3.03	566.12 ±0.95	568.68 ±0.95	0.016 ±0.001	0.016 ±0.001	0.019 ±0.000	0.022 ±0.000
60	0.002 ±0.002	0.002 ±0.002	0.649 ±0.031	0.615 ±0.031	249.29 ±0.66	249.29 ±0.66	587.68 ±1.05	587.71 ±1.02	0.012 ±0.000	0.012 ±0.000	0.020 ±0.000	0.022 ±0.000

D.I.II *Results of Experiments Using the ED-based WFBB-RM Configurations*

Table D.5, Table D.6, Table D.7, and Table D.8 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the ED-based WFBB-RM configurations.

Table D.5. CyberShake workload: effect of λ_{cs} on *P*, *T*, and *O* when using the ED-based WFBB-RM configurations.

λ_{cs} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
1/30	0.002 ±0.001	0.002 ±0.001	0.028 ±0.007	0.035 ±0.008	248.27 ±0.48	248.27 ±0.48	491.37 ±1.37	499.00 ±1.00	0.010 ±0.000	0.010 ±0.000	0.016 ±0.000	0.018 ±0.000
1/22	0.195 ±0.042	0.195 ±0.042	0.131 ±0.022	0.193 ±0.027	293.99 ±2.25	293.99 ±2.25	466.35 ±1.14	482.00 ±1.00	0.016 ±0.001	0.016 ±0.001	0.019 ±0.000	0.022 ±0.000
1/18	1.824 ±0.127	1.824 ±0.127	0.957 ±0.110	1.176 ±0.104	404.99 ±9.00	404.99 ±9.00	474.52 ±2.75	496.00 ±3.00	0.045 ±0.004	0.045 ±0.004	0.025 ±0.001	0.030 ±0.002

Table D.6. CyberShake workload: effect of em_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

em_{max}	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
2	4.173 ±0.112	4.173 ±0.112	4.986 ±0.155	5.428 ±0.155	266.23 ±1.14	266.23 ±1.14	298.85 ±0.73	302.00 ±1.00	0.029 ±0.001	0.029 ±0.001	0.029 ±0.000	0.031 ±0.001
5	0.207 ±0.060	0.207 ±0.060	0.141 ±0.035	0.193 ±0.027	294.52 ±3.03	294.52 ±3.03	467.50 ±1.70	482.00 ±1.00	0.016 ±0.001	0.016 ±0.001	0.020 ±0.000	0.022 ±0.000
10	0.000 ±0.001	0.000 ±0.001	0.016 ±0.005	0.010 ±0.003	301.60 ±1.71	301.60 ±1.71	771.36 ±2.78	787.00 ±3.00	0.012 ±0.000	0.012 ±0.000	0.020 ±0.000	0.024 ±0.000

Table D.7. CyberShake workload: effect of s_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

s_{max} (sec)	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
10000	0.514 ±0.062	0.514 ±0.062	0.317 ±0.037	0.460 ±0.049	357.15 ±2.72	357.15 ±2.72	428.10 ±0.99	455.00 ±1.00	0.022 ±0.001	0.022 ±0.001	0.024 ±0.000	0.027 ±0.000
50000	0.207 ±0.060	0.207 ±0.060	0.141 ±0.035	0.193 ±0.027	294.52 ±3.03	294.52 ±3.03	467.50 ±1.70	482.00 ±1.00	0.016 ±0.001	0.016 ±0.001	0.020 ±0.000	0.022 ±0.000
250000	0.000 ±0.000	0.000 ±0.000	0.014 ±0.005	0.018 ±0.005	231.85 ±0.23	231.85 ±0.23	570.89 ±1.60	573.00 ±1.00	0.008 ±0.000	0.008 ±0.000	0.013 ±0.000	0.015 ±0.000

Table D.8. CyberShake workload: effect of *m* on *P*, *T*, and *O* when using the ED-based WFBB-RM configurations.

m	***P* (%)**				***T* (sec)**				***O* (sec)**			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
40	4.367 ±0.162	4.367 ±0.162	3.454 ±0.203	3.662 ±0.174	706.42 ±23.82	706.42 ±23.82	533.58 ±8.82	550.00 ±8.00	0.088 ±0.008	0.088 ±0.008	0.029 ±0.002	0.034 ±0.003
50	0.207 ±0.060	0.207 ±0.060	0.141 ±0.035	0.193 ±0.027	294.52 ±3.03	294.52 ±3.03	467.50 ±1.70	482.00 ±1.00	0.016 ±0.001	0.016 ±0.001	0.020 ±0.000	0.022 ±0.000
60	0.002 ±0.002	0.002 ±0.002	0.017 ±0.005	0.022 ±0.006	249.29 ±0.66	249.29 ±0.66	493.26 ±1.40	502.00 ±1.00	0.012 ±0.000	0.012 ±0.000	0.020 ±0.000	0.022 ±0.000

D.II. LIGO Workload

This section presents the results of the factor-at-a-time experiments conducted using the LIGO workload. More specifically, the results of the experiments using the PD-based WFBB-RM configurations and the ED-based WFBB-RM configurations are shown in Appendix D.II.I and Appendix D.II.II, respectively.

D.II.I *Results of Experiments Using the PD-based WFBB-RM Configurations*

Table D.9, Table D.10, Table D.11, and Table D.12 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the PD-based WFBB-RM configurations.

Table D.9. LIGO workload: effect of λ_{LG} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

λ_{LG} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
1/265	0.018 ±0.005	0.018 ±0.005	0.418 ±0.059	0.362 ±0.024	1345.90 ±0.64	1345.89 ±0.64	3680.43 ±16.53	3697.90 ±8.33	0.008 ±0.000	0.007 ±0.000	0.014 ±0.001	0.014 ±0.000
1/180	0.107 ±0.014	0.105 ±0.013	1.536 ±0.092	1.444 ±0.096	1466.41 ±4.56	1466.49 ±4.58	3309.03 ±8.24	3326.39 ±8.56	0.009 ±0.000	0.008 ±0.000	0.021 ±0.001	0.021 ±0.000
1/150	1.027 ±0.118	1.047 ±0.114	5.347 ±0.195	5.959 ±0.242	2005.44 ±28.66	2003.05 ±27.76	3588.80 ±25.84	3651.29 ±26.15	0.017 ±0.001	0.015 ±0.001	0.040 ±0.002	0.041 ±0.002

Table D.10. LIGO workload: effect of em_{max} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

em_{max}	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
2	2.439 ±0.139	2.459 ±0.140	11.179 ±0.188	11.333 ±0.213	1457.51 ±4.44	1457.51 ±4.43	1987.27 ±4.66	1992.71 ±4.71	0.012 ±0.000	0.011 ±0.000	0.020 ±0.000	0.021 ±0.000
5	0.107 ±0.014	0.105 ±0.013	1.536 ±0.092	1.444 ±0.096	1466.41 ±4.56	1466.49 ±4.58	3309.03 ±8.24	3326.39 ±8.56	0.009 ±0.000	0.008 ±0.000	0.021 ±0.001	0.021 ±0.000
10	0.041 ±0.011	0.042 ±0.008	0.197 ±0.021	0.180 ±0.018	1458.44 ±6.18	1463.48 ±4.62	5212.01 ±19.41	5243.74 ±16.92	0.009 ±0.000	0.008 ±0.000	0.023 ±0.000	0.027 ±0.000

Table D.11. LIGO workload: effect of s_{max} on P, T, and O when using the PD-based WFBB-RM configurations.

s_{max} (sec)	P (%)				T (sec)				O (sec)			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
10000	0.099 ±0.012	0.101 ±0.012	1.395 ±0.077	1.305 ±0.077	1450.10 ±3.34	1450.17 ±3.35	3257.78 ±8.26	3278.96 ±8.89	0.009 ±0.000	0.008 ±0.000	0.020 ±0.000	0.020 ±0.000
50000	0.107 ±0.014	0.105 ±0.013	1.536 ±0.092	1.444 ±0.096	1466.41 ±4.56	1466.49 ±4.58	3309.03 ±8.24	3326.39 ±8.56	0.009 ±0.000	0.008 ±0.000	0.021 ±0.001	0.021 ±0.000
250000	0.092 ±0.012	0.081 ±0.011	1.147 ±0.076	1.109 ±0.077	1440.81 ±4.73	1427.49 ±4.20	3448.23 ±8.57	3464.41 ±7.99	0.010 ±0.000	0.008 ±0.000	0.018 ±0.000	0.018 ±0.000

Table D.12. LIGO workload: effect of m on P, T, and O when using the PD-based WFBB-RM configurations.

m	P (%)				T (sec)				O (sec)			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
40	4.113 ±0.266	4.136 ±0.285	7.862 ±0.229	8.124 ±0.254	3210.01 ±124.66	3210.49 ±125.21	4680.11 ±119.89	4769.01 ±125.60	0.034 ±0.003	0.033 ±0.005	0.062 ±0.004	0.062 ±0.004
50	0.107 ±0.014	0.105 ±0.013	1.536 ±0.092	1.444 ±0.096	1466.41 ±4.56	1466.49 ±4.58	3309.03 ±8.24	3326.39 ±8.56	0.009 ±0.000	0.008 ±0.000	0.021 ±0.001	0.021 ±0.000
60	0.028 ±0.006	0.028 ±0.006	0.621 ±0.040	0.582 ±0.039	1360.27 ±1.13	1360.27 ±1.13	3509.39 ±8.43	3519.75 ±8.53	0.010 ±0.000	0.009 ±0.000	0.019 ±0.000	0.019 ±0.000

D.II.II *Results of Experiments Using the ED-based WFBB-RM Configurations*

Table D.13, Table D.14, Table D.15, and Table D.16 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the ED-based WFBB-RM configurations.

Table D.13. LIGO workload: effect of λ_{LG} on *P*, *T*, and *O* when using the ED-based WFBB-RM configurations.

λ_{LG} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
1/265	0.018 ±0.005	0.018 ±0.005	0.108 ±0.019	0.104 ±0.015	1345.90 ±0.64	1345.90 ±0.64	3673.14 ±10.26	3662.84 ±7.09	0.008 ±0.000	0.007 ±0.000	0.013 ±0.000	0.014 ±0.000
1/180	0.107 ±0.014	0.108 ±0.015	0.916 ±0.113	0.802 ±0.067	1466.33 ±4.56	1466.58 ±4.60	3277.91 ±10.38	3279.22 ±6.77	0.009 ±0.000	0.008 ±0.000	0.019 ±0.000	0.021 ±0.000
1/150	1.057 ±0.122	0.989 ±0.113	5.423 ±0.341	5.146 ±0.235	2005.59 ±28.09	2004.65 ±28.02	3647.43 ±50.91	3612.06 ±28.17	0.016 ±0.001	0.015 ±0.001	0.040 ±0.002	0.041 ±0.002

Table D.14. LIGO workload: effect of em_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

em_{max}	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
2	2.434 ±0.142	2.436 ±0.141	9.838 ±0.272	10.006 ±0.189	1457.46 ±4.42	1457.71 ±4.44	1966.23 ±6.10	1979.70 ±4.58	0.011 ±0.000	0.010 ±0.000	0.020 ±0.000	0.021 ±0.000
5	0.107 ±0.014	0.108 ±0.015	0.916 ±0.113	0.802 ±0.067	1466.33 ±4.56	1466.58 ±4.60	3277.91 ±10.38	3279.22 ±6.77	0.009 ±0.000	0.008 ±0.000	0.019 ±0.000	0.021 ±0.000
10	0.043 ±0.008	0.044 ±0.008	0.063 ±0.013	0.082 ±0.011	1463.46 ±4.61	1463.44 ±4.62	5258.83 ±21.26	5242.99 ±14.51	0.008 ±0.000	0.008 ±0.000	0.023 ±0.000	0.024 ±0.000

Table D.15. LIGO workload: effect of s_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

s_{max} (sec)	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
10000	0.103 ±0.012	0.102 ±0.012	0.518 ±0.064	0.629 ±0.054	1450.17 ±3.35	1450.16 ±3.35	2987.98 ±8.02	3227.28 ±6.62	0.009 ±0.000	0.008 ±0.000	0.020 ±0.000	0.020 ±0.000
50000	0.107 ±0.014	0.108 ±0.015	0.916 ±0.113	0.802 ±0.067	1466.33 ±4.56	1466.58 ±4.60	3277.91 ±10.38	3279.22 ±6.77	0.009 ±0.000	0.008 ±0.000	0.019 ±0.000	0.021 ±0.000
250000	0.080 ±0.010	0.082 ±0.011	0.699 ±0.065	0.602 ±0.062	1427.32 ±4.14	1427.44 ±4.16	3421.39 ±7.05	3420.67 ±7.27	0.009 ±0.000	0.008 ±0.000	0.018 ±0.000	0.170 ±0.000

Table D.16. LIGO workload: effect of m on P, T, and O when using the ED-based WFBB-RM configurations.

m	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
40	4.144 ±0.269	4.093 ±0.268	7.901 ±0.227	8.006 ±0.240	3217.69 ±125.83	3214.09 ±127.07	4901.42 ±127.63	4769.89 ±123.16	0.032 ±0.003	0.031 ±0.004	0.062 ±0.004	0.061 ±0.040
50	0.107 ±0.014	0.108 ±0.015	0.916 ±0.113	0.802 ±0.067	1466.33 ±4.56	1466.58 ±4.60	3277.91 ±10.38	3279.22 ±6.77	0.009 ±0.000	0.008 ±0.000	0.019 ±0.000	0.021 ±0.000
60	0.028 ±0.006	0.028 ±0.006	0.179 ±0.019	0.160 ±0.018	1360.27 ±1.13	1360.27 ±1.13	3461.67 ±7.81	3462.32 ±7.60	0.010 ±0.000	0.009 ±0.000	0.019 ±0.000	0.020 ±0.000

D.III. Genome Workload

The results of the factor-at-a-time experiments conducted using the Genome workload are presented in this section. More specifically, the results of the experiments using the PD-based WFBB-RM configurations and the ED-based WFBB-RM configurations are shown in Appendix D.III.I and Appendix D.III.II, respectively.

D.III.I *Results of Experiments Using the PD-based WFBB-RM Configurations*

Table D.17, Table D.18, Table D.19, and Table D.20 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the PD-based WFBB-RM configurations.

Table D.17. Genome workload: effect of λ_{GN} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

λ_{GN} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
1/3205	0.013 ±0.004	0.013 ±0.004	0.122 ±0.022	0.124 ±0.024	17544 ±927	17544 ±927	48911 ±2234	48969 ±2242	0.008 ±0.000	0.008 ±0.000	0.010 ±0.000	0.010 ±0.000
1/2290	0.066 ±0.013	0.066 ±0.013	0.487 ±0.095	0.614 ±0.119	17963 ±1007	17963 ±1007	45666 ±1958	45934 ±2000	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
1/1800	1.432 ±0.451	1.408 ±0.452	2.600 ±0.567	2.918 ±0.605	52312 ±12915	52198 ±12897	80318 ±14218	80645 ±14207	0.048 ±0.015	0.048 ±0.015	0.044 ±0.019	0.045 ±0.020

Table D.18. Genome workload: effect of em_{max} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

em_{max}	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
2	0.490 ±0.123	0.489 ±0.123	7.792 ±1.342	8.108 ±1.421	17933 ±1001	17933 ±1001	26325 ±1446	26437 ±1465	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
5	0.066 ±0.013	0.066 ±0.013	0.487 ±0.095	0.614 ±0.119	17963 ±1007	17963 ±1007	45666 ±1958	45934 ±2000	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
10	0.028 ±0.006	0.028 ±0.006	0.081 ±0.013	0.097 ±0.015	17963 ±1007	17963 ±1007	74866 ±2616	75009 ±2646	0.007 ±0.000	0.007 ±0.000	0.013 ±0.001	0.013 ±0.001

Table D.19. Genome workload: effect of s_{max} on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

s_{max} (sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
10000	0.037 ±0.009	0.037 ±0.009	0.431 ±0.084	0.574 ±0.114	17693 ±959	17693 ±959	45729 ±1942	46052 ±1994	0.007 ±0.000	0.007 ±0.000	0.012 ±0.001	0.011 ±0.001
50000	0.066 ±0.013	0.066 ±0.013	0.487 ±0.095	0.614 ±0.119	17963 ±1007	17963 ±1007	45666 ±1958	45934 ±2000	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
250000	0.080 ±0.015	0.080 ±0.015	0.592 ±0.125	0.687 ±0.153	18171 ±1049	18172 ±1049	45962 ±2009	46194 ±2046	0.007 ±0.000	0.007 ±0.000	0.012 ±0.001	0.011 ±0.001

Table D.20. Genome workload: effect of *m* on *P*, *T*, and *O* when using the PD-based WFBB-RM configurations.

m	***P* (%)**				***T* (sec)**				***O* (sec)**			
	PD-SL-TSP1	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**	**PD-SL-TSP1**	**PD-TL-TSP1**	**PD-SL-TSP2**	**PD-TL-TSP2**
40	1.286 ±0.403	1.322 ±0.417	2.461 ±0.510	2.741 ±0.544	52320 ±13743	52266 ±13692	79537 ±14854	79726 ±14708	0.032 ±0.011	0.032 ±0.011	0.031 ±0.014	0.032 ±0.015
50	0.066 ±0.013	0.066 ±0.013	0.487 ±0.095	0.614 ±0.119	17963 ±1007	17963 ±1007	45666 ±1958	45934 ±2000	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
60	0.016 ±0.004	0.016 ±0.004	0.166 ±0.029	0.186 ±0.034	17583 ±935	17583 ±935	47591 ±2080	47682 ±2095	0.009 ±0.000	0.009 ±0.000	0.012 ±0.000	0.012 ±0.000

D.III.II *Results of Experiments Using the ED-based WFBB-RM Configurations*

Table D.21, Table D.22, Table D.23, and Table D.24 present the effect of job arrival rate, effect of earliest start time of jobs, effect of job deadlines, and effect of the number of resources, respectively, when using the ED-based WFBB-RM configurations.

Table D.21. Genome workload: effect of λ_{GN} on *P*, *T*, and *O* when using the ED-based WFBB-RM configurations.

λ_{GN} (jobs/sec)	***P* (%)**				***T* (sec)**				***O* (sec)**			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
1/3205	0.013 ±0.004	0.013 ±0.004	0.052 ±0.010	0.059 ±0.012	17544 ±927	17544 ±927	48358 ±2155	48462 ±2170	0.008 ±0.000	0.008 ±0.000	0.010 ±0.000	0.009 ±0.000
1/2290	0.066 ±0.013	0.066 ±0.013	0.188 ±0.039	0.232 ±0.045	17963 ±1007	17963 ±1007	44082 ±1744	44471 ±1795	0.008 ±0.000	0.008 ±0.000	0.012 ±0.001	0.011 ±0.000
1/1800	1.395 ±0.442	1.477 ±0.470	2.195 ±0.577	2.309 ±0.586	52472 ±13003	52134 ±12926	78558 ±14192	78476 ±14011	0.041 ±0.016	0.041 ±0.021	0.045 ±0.019	0.043 ±0.019

Table D.22. Genome workload: effect of em_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

em_{max}	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
2	0.489 ±0.123	0.492 ±0.125	6.405 ±1.193	6.645 ±1.285	17933 ±1001	17933 ±1001	26088 ±1408	26257 ±1438	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
5	0.066 ±0.013	0.066 ±0.013	0.188 ±0.039	0.188 ±0.039	17963 ±1007	17963 ±1007	44082 ±1744	44082 ±1744	0.008 ±0.000	0.008 ±0.000	0.012 ±0.001	0.012 ±0.001
10	0.028 ±0.006	0.028 ±0.006	0.050 ±0.008	0.064 ±0.010	17963 ±1007	17963 ±1007	72259 ±2303	72922 ±2383	0.008 ±0.000	0.008 ±0.000	0.014 ±0.001	0.014 ±0.001

Table D.23. Genome workload: effect of s_{max} on P, T, and O when using the ED-based WFBB-RM configurations.

s_{max} (sec)	P (%)				T (sec)				O (sec)			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
10000	0.037 ±0.009	0.037 ±0.009	0.170 ±0.037	0.216 ±0.044	17693 ±959	17693 ±959	44181 ±1729	44524 ±1777	0.008 ±0.000	0.008 ±0.000	0.011 ±0.001	0.011 ±0.001
50000	0.066 ±0.013	0.066 ±0.013	0.188 ±0.039	0.188 ±0.039	17963 ±1007	17963 ±1007	44082 ±1744	44082 ±1744	0.008 ±0.000	0.008 ±0.000	0.012 ±0.001	0.012 ±0.001
250000	0.080 ±0.015	0.080 ±0.015	0.290 ±0.082	0.302 ±0.075	18171 ±1049	18172 ±1049	44399 ±1806	44749 ±1850	0.008 ±0.000	0.008 ±0.000	0.012 ±0.001	0.012 ±0.001

Table D.24. Genome workload: effect of *m* on *P*, *T*, and *O* when using the ED-based WFBB-RM configurations.

m	***P* (%)**				***T* (sec)**				***O* (sec)**			
	ED-SL-TSP1	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**	**ED-SL-TSP1**	**ED-TL-TSP1**	**ED-SL-TSP2**	**ED-TL-TSP2**
40	1.305 ±0.417	1.310 ±0.414	2.147 ±0.517	2.220 ±0.517	52106 ±13597	52128 ±13629	78411 ±14887	78278 ±14615	0.035 ±0.012	0.028 ±0.015	0.031 ±0.014	0.032 ±0.014
50	0.066 ±0.013	0.066 ±0.013	0.188 ±0.039	0.188 ±0.039	17963 ±1007	17963 ±1007	44082 ±1744	44082 ±1744	0.008 ±0.000	0.008 ±0.000	0.012 ±0.001	0.012 ±0.001
60	0.016 ±0.004	0.016 ±0.004	0.057 ±0.011	0.065 ±0.012	17583 ±935	17583 ±935	46641 ±1937	46784 ±1958	0.010 ±0.000	0.010 ±0.000	0.012 ±0.000	0.012 ±0.000

D.IV. Comparison of WFBB-RM and MRCP-RM

This section presents the additional results of the factor-at-a-time experiments conducted to compare the performances of WFBB-RM and MRCP-RM when processing an open stream of MapReduce jobs with SLAs (refer to Section 7.6).

D.IV.I *Effect of Task Execution Times*

Figure D.1 and Figure D.2 present the performance of WFBB-RM and MRCP-RM in terms of P, T, and O when me_{max} is varied.

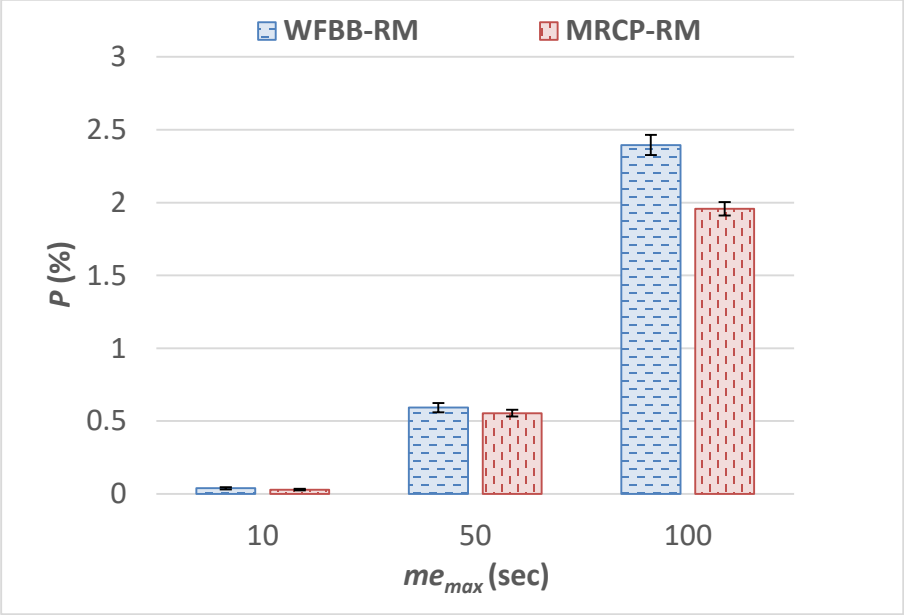

Figure D.1. WFBB-RM vs MRCP-RM: effect of me_{max} on P.

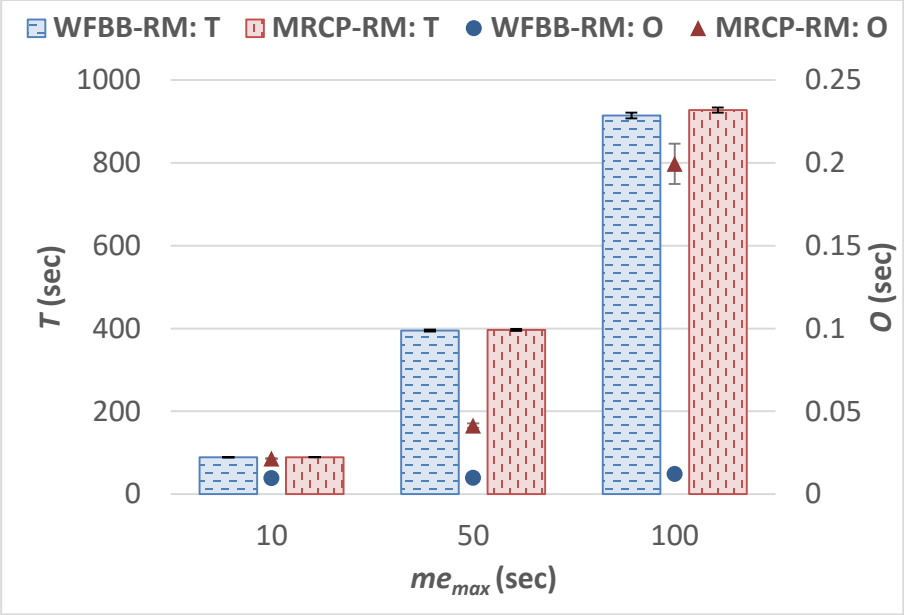

Figure D.2. WFBB-RM vs MRCP-RM: effect of me_{max} on T and O.

D.IV.II *Effect of Earliest Start Time of Jobs*

The performance of WFBB-RM and MRCP-RM in terms of P, T, and O when s_{max} is varied is presented in Figure D.3 and Figure D.4.

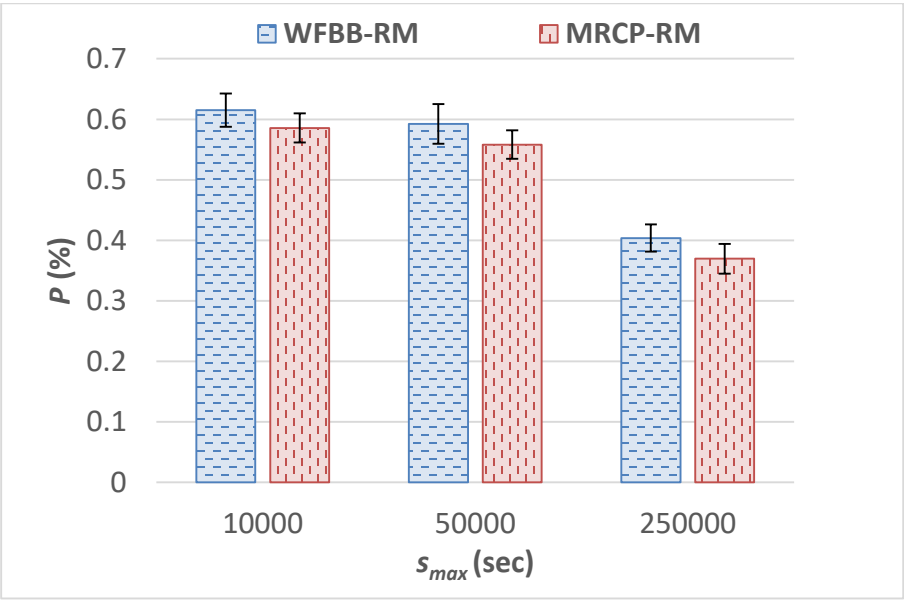

Figure D.3. WFBB-RM vs MRCP-RM: effect of s_{max} on P.

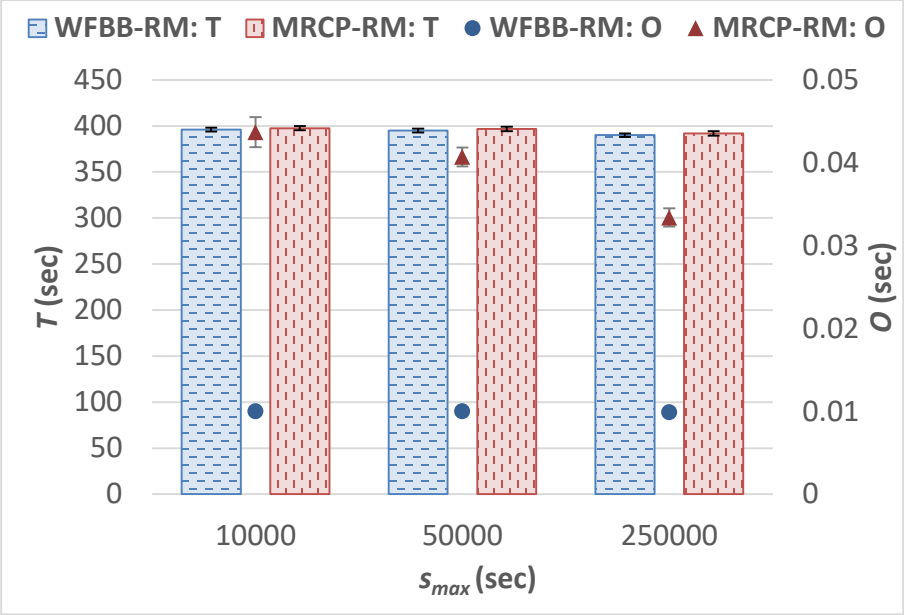

Figure D.4. WFBB-RM vs MRCP-RM: effect of s_{max} on T and O.

D.IV.III *Effect of Job Deadlines*

Figure D.5 and Figure D.6 present the performance of WFBB-RM and MRCP-RM in terms of P, T, and O when em_{max} is varied.

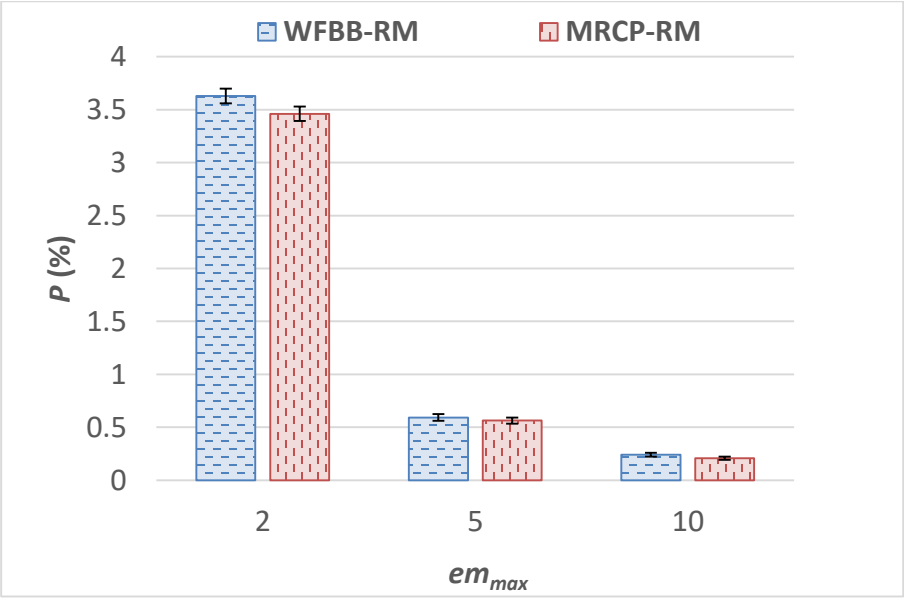

Figure D.5. WFBB-RM vs MRCP-RM: effect of em_{max} on P.

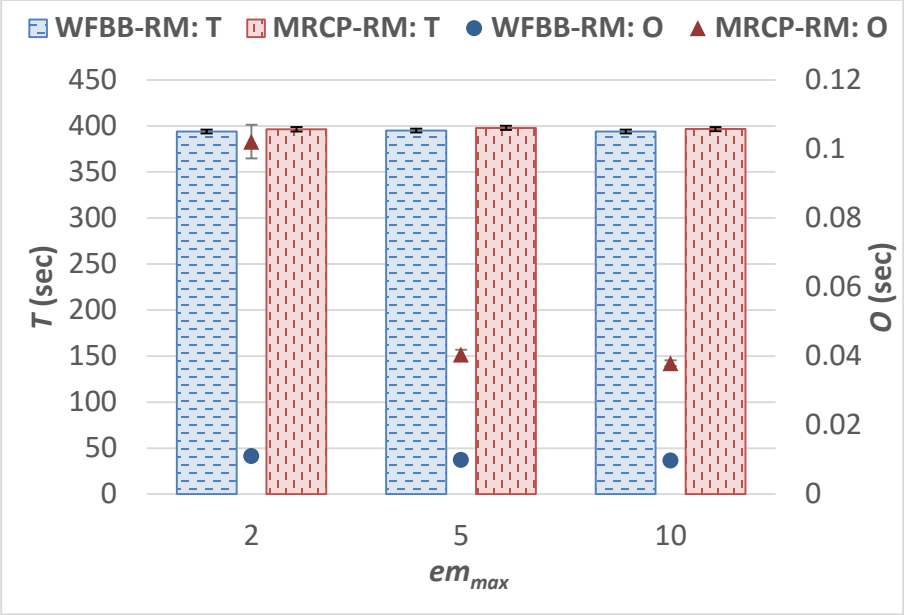

Figure D.6. WFBB-RM vs MRCP-RM: effect of em_{max} on T and O.