Engineering Resource Management Middleware for Optimizing the Performance of Clouds Processing MapReduce Jobs with Deadlines

Norman Lim
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
nlim@sce.carleton.ca

Shikharesh Majumdar
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
majumdar@sce.carleton.ca

Peter Ashwood-Smith
Huawei Technologies Canada
Kanata, ON, Canada
Peter.AshwoodSmith@huawei.com

[Industrial and Experience Paper]

ABSTRACT

This paper focuses on devising efficient resource management techniques used by the resource management middleware in clouds that handle MapReduce jobs with end-to-end service level agreements (SLAs) comprising an earliest start time, execution time, and a deadline. This research and development work, performed in collaboration with our industrial partner, presents the formulation of the matchmaking and scheduling problem for MapReduce jobs as an optimization problem using: Mixed Integer Linear Programming (MILP) and Constraint Programming (CP) techniques. In addition to the formulations devised, our experience in implementing the MILP and CP models using various open source as well as commercial software packages is described. Furthermore, a performance evaluation of the different approaches used to implement the formulations is conducted using a variety of different workloads.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems. C.4 [**Performance of Systems**]: *performance attributes, modeling techniques*.

Keywords

Resource management on clouds, MapReduce with deadlines, Optimization, Mixed integer linear programming (MILP), Constraint programming (CP).

1. INTRODUCTION

Cloud computing, which concerns improving the way Information Technology (IT) is managed and consumed is receiving a great deal of interest from researchers and practioners from academia and industry. Cloud computing makes computational (hardware and software) resources accessible as scalable and on-demand services over a network such as the Internet [1]. To accomplish this goal, the cloud computing paradigm employs a wide-range of concepts and technologies such as virtualization, service-orientation, elasticity, scalability, and pay-as-you-go. Using the virtualization technology, cloud computing is able to 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). IaaS delivers basic computational resources (virtual machines) as an on-demand service whereas PaaS offers a higher-level service (e.g. application framework with development tools) where consumers can create and deploy their own scalable Web applications without having to invest in and maintain their own physical infrastructure. Lastly, SaaS provides consumers with complete end-user Web applications. Communication and social applications such as Customer Relationship Management (CRM) systems, email, and Facebook are examples of SaaS. Along with an on-demand service-oriented model, cloud computing also offers *scalability*, *elasticity*, and *pay-as-you-go* features. The scalability and elasticity characteristics of the cloud provide the ability to grow or shrink the number of resources allocated to a consumer's request dynamically with time. With the pay-as-you-go model, consumers can lease resources on-demand from the service provider and pay only for the time the resources are used.

In addition to researchers and service consumers, cloud computing that is based on resources acquired on demand is generating a great deal of interest among service providers and system builders as well. Cloud service providers typically own a large pool of resources that include computing, storage, and communication resources. Effective resource management strategies and performance optimization techniques need to be developed for harnessing the power of the underlying resource pool. The important operations performed by a resource manager deployed in the resource management middleware for a cloud include: *matchmaking* and *scheduling*. When a request arrives, the resource manager invokes a *matchmaking* algorithm that selects the resource or resources (from a given a pool of resources) to be allocated to the request. Once a number of requests get allocated to a specific resource, a *scheduling* algorithm is used to determine the order in which these requests are to be executed. Both matchmaking and scheduling are well known as computationally hard problems because they need to satisfy a user's requirements for a quality of service that is often captured in a service level agreement (SLA); while also achieving the desired system objectives for the service providers, such as generating a high resource utilization and adequate revenue. Matchmaking and scheduling decisions can be made in one joint step (see [2] and [3] for example). The resource management technique presented in this paper makes such a joint decision on matchmaking and scheduling. Note that in this paper the term *output schedule* is used to define the task to resource mapping, and when each task runs on their assigned resource.

Both performance optimization and performance modeling are important components of performance engineering. This

ICPE'14, March 22–26 2014, Dublin, Ireland

http://dx.doi.org/10.1145/2568088.2576796

research concerns engineering resource management middleware that has two primary objectives: reducing the overhead of making resource management decisions and making decisions that achieve high system performance. This paper focuses on the investigation of techniques for management of resources on clouds in which the workload includes requests characterized by an end-to-end SLA [1] that comprises an earliest start time, execution time, and (soft) deadline [4] specified by the user. On systems with soft deadlines, jobs are permitted to miss their deadlines; however, the desired system objective is to minimize the number of jobs that do miss their deadlines. In addition, these requests also require *multiple stages of execution* with potentially different resources used in each stage. Considering these types of requests poses a new challenge to the resource management problem. Most of the works on resource management on clouds for workloads characterized by SLAs have only considered requests that require a single resource (i.e. single stage of execution). A scenario that involves reserving multiple resources simultaneously is important in the context of applications that require multiple system components, and can also arise from workflows that require the interaction of multiple applications that run on different resources. In this context, the workloads that are considered in this paper consist of *MapReduce* jobs.

MapReduce is a programming model that is characterized by multiple stages of execution and requires that the user define two functions [5]: a *map function* and a *reduce function*. Google proposed the use of MapReduce for processing large amounts (e.g. terabytes) of raw data in a distributed (parallel) manner to generate or derive more meaningful data [5]. The map function generates a set of intermediate key/value pairs from a set of input key/value pairs. These intermediate key/value pairs are grouped together and then passed to the reduce function where the values with identical keys are merged. A typical MapReduce job consists of a set of map tasks and a set of reduce tasks. The reduce tasks generally do not start executing until all map tasks are completed. Many computations can be expressed using MapReduce. For example, a MapReduce application can be deployed to count the number of URL accesses on a web server [5]. In this application, the map function processes the web server logs and produces a data set with an intermediate key/value pair of the form: {URL, 1}. This new data set is then processed by the reduce function, which sums all the values with identical keys to emit a new data set with the following key/value pair: {URL, total count}.

A popular implementation of MapReduce is Apache Hadoop [6], which is used by many companies and institutions for a variety of applications such as data processing (e.g., sorting, indexing, and grouping), data analysis, data mining (e.g. web crawling), machine learning, and scientific research (e.g. bioinformatics) [7]. In all these cases, there may be situations where a batch of MapReduce jobs needs to be executed either on a private cluster, or a cloud (such as Amazon EC2). On both systems, matchmaking and scheduling a batch of MapReduce jobs needs to be performed by the resource management middleware. Completing each job in the batch within a specific period of time (characterized by a deadline) is often a user requirement [4][8]. Resource management on such an environment is the focus of attention in this paper. Similar systems that map and schedule a batch of MapReduce jobs are investigated in [8], [9], and [10].

The goal of our research, performed in collaboration with Huawei Technologies, Canada, is to devise a cloud resource manager that can effectively perform matchmaking and scheduling of MapReduce jobs each of which is characterized by an end-to-end SLA comprising an earliest start time, execution time, and a deadline specified by the user. In this paper, we focus on describing our investigation and experiences with using various techniques and technologies to formulate and solve the matchmaking and scheduling problem. Figure 1 displays the three different approaches that are used for solving the matchmaking and scheduling problem. As shown in Figure 1, we formulate the matchmaking and scheduling problem as an optimization problem, and solve the matchmaking and scheduling problem jointly. More specifically, we describe our investigation/ experience in formulating the problem using: (1) *Mixed Integer Linear Programming* [11][12] (MILP), and (2) *Constraint Programming* [13] (CP) techniques. Both MILP and CP are well-known theoretical techniques that can solve optimization problems and find *optimal* solutions. See Section 2 for a further discussion on MILP and CP. Various implementations of our solutions based on MILP and CP using different software packages are considered: *Approach 1*: MILP model implemented and solved using LINGO [14] (commercial software); *Approach 2*: CP model implemented using MiniZinc/FlatZinc [15] and solved using Gecode [16] (both open source software); and *Approach 3*: CP model implemented and solved using IBM ILOG CPLEX Optimization Studio (CPLEX) [17] (commercial software).This paper is motivated by issues such as:

- How to employ the existing theory on MILP and CP for devising efficient resource management algorithms that minimize the number of jobs missing their deadlines on a closed system subjected to a batch workload comprising MapReduce jobs with deadlines.
- Development of efficient implementations of the algorithms using Commercial-Off-The-Shelf (COTS) packages that produce an acceptable system overhead accrued during the execution of the resource management algorithms.
- Getting an understanding of the relationship between the size of the workload and system performance.

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* (see Figure 1). The MILP/CP solver program corresponding to the embodiment of an approach is executed on a desktop PC (described in Section 5). As captured in Figure 1 the *output schedule* (the mapping of tasks on resources and their assigned start times), along with the time required to complete the execution of the batch of MapReduce jobs, and the number of jobs missing their deadlines are obtained as an output at the end of a given experiment. The processing time required by the solver to produce the output is also measured by using the solver's built-in timing utilities. A performance evaluation was conducted to compare the three approaches using various system and workload parameters. Our goal is to determine which of these approaches is able to solve the matchmaking and scheduling problem for MapReduce jobs efficiently, and understand the trade-off between processing time and the quality of the output schedule in terms of the number of jobs missing their deadlines, and the completion time of the workload. The main contributions of this paper include:

- Devising a technique for generating an optimal solution that minimizes the number of jobs missing their deadlines. The formulation of two models, one using MILP and one using CP, for achieving the optimal solution for matchmaking and scheduling MapReduce jobs with SLAs are presented.
  - A comparison of MILP and CP, focusing on the differences in using these two techniques for achieving the resource management technique.

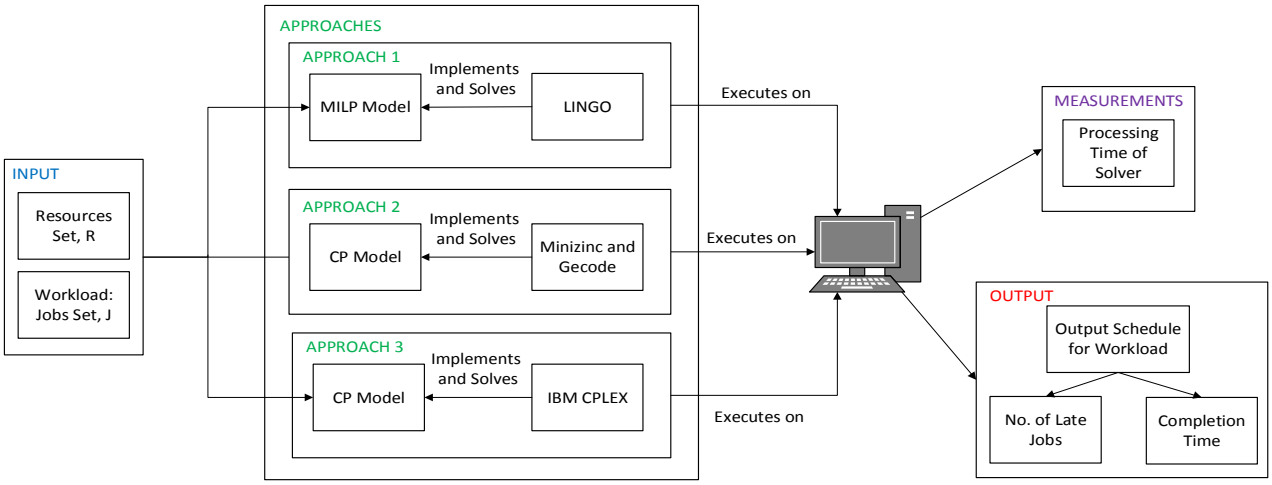

Figure 1. Overview of our approach.

- A discussion of the three approaches used to implement the MILP and CP models is presented.
- A performance evaluation of the three approaches using a variety of different system and workload parameters is presented. Insights gained into system behavior from the results of the performance evaluation are described.

The results of this research will be useful to researchers, designers, and users of the resource management middleware, including system developers and cloud service providers.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to MILP and CP, and presents related work. Section 3 presents the problem description, and the formulations of the MILP and CP models. The design and implementation of the MILP and CP models using the three software packages are discussed in Section 4. In Section 5, the results of the performance evaluation, and a comparison of the performance of the three approaches are presented. Lastly, Section 6 concludes the paper and provides directions for future work.

2. BACKGROUND AND RELATED WORK

MILP [11] and CP [13] are well-known techniques that are used to solve optimization problems, and are capable of finding *optimal* solutions with regards to maximizing or minimizing an objective function. Both techniques have the same general modeling structure. There are a set of *decision variables* that need to be assigned values that ensure an *objective function* is optimized (maximized or minimized) subject to *constraints*—conditions that cannot be violated. In addition, both MILP and CP have been shown to be effective in solving planning and scheduling problems, such as the traditional job shop scheduling problem [3]. MILP is a subfield of mathematical programming (MP) (also called 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 [12]. The theoretical basis for MILP and mathematical optimization in general is numerical algebra [18]. To solve MILP models, techniques such as cutting-planes (constraint relaxations) and Branch and Bound are used.

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 [13]. This theoretical foundation for CP is different than the theoretical foundation for MP techniques, such as MILP [18]. Unlike MILP models, CP models natively support a variety of arithmetic operators and logical constraints such as integer division, and the 'implies' constraint [19]. To formulate logical constraints in a MILP model, the 'big-M' formulation technique [11] is typically used [19]. CP also defines a general set of specialized constraints, called *global constraints* that model frequently used patterns seen in optimization problems [20]. For example, one such constraint is the *cumulative* constraint which is used in scheduling problems to ensure that the capacity of each resource is not violated at any point in time.

The main limitation of CP models is that, natively, the decision variables can only be discrete (i.e. integer or Boolean) [13], whereas MP models support both discrete and continuous decision variables. The theoretical basis for solving MP models is numerical algebra; in contrast, for CP models the theoretical basis is logical inference including logic and graph theory. Search algorithms, including back-tracking and local search [13], 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 to 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.

A significant body of knowledge exists in the area of resource management on grids and clouds. More recently, researchers have also started investigating the problem of scheduling and matchmaking MapReduce jobs with deadlines. Due to space limitations, a representative set of work is presented. Development of a resource management middleware for clouds that is able to make smart and global decisions for achieving high system performance is being considered in OpenStack [21], which is collaborative open-source cloud software project. The authors of [8], propose a Deadline Constraint Scheduler for Hadoop [6] to handle jobs with deadlines. A job execution cost model that considers parameters including the execution time of map and reduce tasks, and input data sizes is developed. Dong et al. [9] focus on the scheduling of workloads comprising of MapReduce jobs with deadlines (real-time jobs) and jobs with no deadlines (non-real-time jobs). They integrate techniques such as Tasks Forward Scheduling (TFS) and Approximately Uniform Minimum Degree of Parallelism (AUMD) into the existing Hadoop scheduler to form a two-level scheduler that is capable of scheduling both real-time and non-real-time jobs. In [4], Verma et al. propose two resource allocation policies based on the earliest deadline first (EDF) strategy for Hadoop. The first policy is *MinEDF*, which allocates the minimum number of task slots required for completing a job before its deadline. The second policy, called *MinEDF-WC*, enhances MinEDF by adding the

ability to dynamically allocate and de-allocate resources (task slots) from active jobs according to demand.

Similar to our research, the works discussed in [4], [8], and [9] investigate matchmaking and scheduling MapReduce jobs with deadlines. However, these works propose heuristic-based schedulers whereas our work describes using optimization techniques that can generate optimal solutions. These works also do not consider jobs characterized with earliest start times, which can be important in the context of advance reservation requests. In addition, the focus of [4], [8], and [9], is on improving the Hadoop [6] scheduler, which is an open source framework that implements the MapReduce programming model. The research described in this paper describes solutions for the general matchmaking and scheduling problem for MapReduce jobs with deadlines.

3. PROBLEM DESCRIPTION AND MODEL FORMULATIONS

This section describes the modeling of matchmaking and scheduling MapReduce jobs with end-to-end SLAs comprising an earliest start time, execution time, and a deadline. First, a model of the MapReduce matchmaking and scheduling problem is presented (Input box of Figure 1). The formulations of the MILP and CP models are then described in Sections 3.1 and 3.2, respectively. A general matchmaking and scheduling problem requires two components for input data: a *workload* component, and a *system* component [2]. The workload data component outlines the characteristics of the jobs, whereas the system data component defines the attributes of the resources that the jobs will be executed on.

In our model, each MapReduce job j in the set of jobs, J, needs to mapped and scheduled on a cloud environment with m resources (or computing nodes), which is represented by a set $R = \{r_1, r_2, \dots, r_m\}$. Each resource r in the set R has: (1) a map task capacity (or number of map slots),c_r^{mp}, and (2) a reduce task capacity (or number of reduce slots),c_r^{rd}. The map and reduce task capacities specify the number of map and reduce tasks, respectively, that each of the resources can execute in parallel at a point in time. In addition, the map task slots are independent from the reduce tasks slots, which means that a map task can be run at the same time that a reduce task is executing.

The workload comprises a set (batch) of MapReduce jobs to schedule, $J = \{j_1, j_2, \dots, j_n\}$ where n is the number of jobs in the set. Each job j in the set J has the following:

- A set of map tasks $T_j^{mp} = \{t_{j,1}^{mp}, t_{j,2}^{mp}, \dots t_{j,k_j^{mp}}^{mp}\}$ where k_j^{mp} denotes the number of map tasks that are in job j.
- A set of reduce tasks $T_j^{rd} = \{t_{j,1}^{rd}, t_{j,2}^{rd}, \dots t_{j,k_j^{rd}}^{rd}\}$ where k_j^{rd} denotes the number of reduce tasks in job j.
- A set $T_j = \{T_j^{mp}, T_j^{rd}\}$ contains all tasks for job j.
- Earliest start time for the job, s_j
- Deadline for the job, d_j, by which the job should be completed (i.e. soft deadline).

Each task t in T_j has the following attributes: (1) a required execution time,e_t, and (2) a resource capacity requirement, q_t. Note that typical map and reduce tasks only require executing on one resource slot [4]. As such, q_t is typically set to one. All the tasks of all jobs are placed in a master set T.

The requirements for mapping and scheduling the set of jobs J on to the set of resources R are summarized. Each task t in T_j can only be scheduled to start at, or after job j's earliest start time, s_j. Secondly, each task t in T can only be mapped to a single resource r where t executes on r for e_t time units. Map tasks and reduce tasks can be executed in parallel, however, all the map tasks have to be completed before the reduce tasks 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 it can handle). The system objective for the resource manager (objective function) is to minimize the number of jobs that miss their deadlines (i.e. minimize the number of late jobs).

3.1 Formulation of the MILP Model

The MILP model uses a time-indexed formulation [22], which is a commonly used model for formulating scheduling problems that considers discrete time (i.e. integer values for time). The discrete time values are contained in a set I called the *time range*. 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 5.1 seconds, the time can be converted into a discrete time value by changing it to 5100 milliseconds. In some cases, if the length of times are very different (e.g., 0.1s versus 10^3s), it may not best to change the unit of time because the converted values can be quite large (e.g. 0.1s becomes 100ms and 10^3s becomes 10^6ms). In these cases, it may be more appropriate to round the non-discrete time values to the nearest higher integer. For instance, the 0.1s can be rounded up to 1s.

Table 1 presents the formulation of the MILP model. Recall that the input of the MILP model comprises a set of resources R on which to execute a set of jobs J, and that a set T contains all tasks of all jobs in J. The following decision variables are defined in the MILP model:

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

Table 1: Formulation of the MILP Model

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

$$\sum_{i \in I} \sum_{r \in R} x_{tri} = 1 \quad \forall\, t \in T \tag{1a}$$

$$\left((i|\, x_{tri} = 1) \geq s_j \quad \forall t \in T_j^{mp},\, \forall r \in R, \forall i \in I \right) \quad \forall j \in J \tag{2a}$$

$$\left(i|\, x_{tri} = 1 \geq \max_{t' \in T_j^{mp}, r' \in R, i' \in I} ((i'|x_{t'r'i'} = 1) + e_{t'}) \quad \forall t \in T_j^{rd},\, \forall r \in R, \forall i \in I \right) \forall j \in J \tag{3a}$$

$$\left(N_j d_j \geq \max_{t \in T_j^{rd}, r \in R, i \in I} \left((i|\, x_{tri} = 1) + e_t \right) - d_j \right) \quad \forall j \in J \tag{4a}$$

$$\sum_{t \in T^{mp}} \sum_{i' \in I^*_{tri}} x_{tri'} q_t \leq c_r^{mp} \quad \forall r \in R,\, \forall i \in I \tag{5a}$$

$$\text{where } I^*_{tri} = \{i' | i - e_t < i' \leq i\}$$

Same as (5) but for reduce tasks. (6a)

$$x_{tri} \in \{0,\, 1\} \qquad \forall\, t \in T, \quad \forall r \in R,\, \forall i \in I \tag{7a}$$

$$N_j \in \{0,\, 1\} \qquad \forall\, j \in J \tag{8a}$$

$$i \in \mathbb{Z} \tag{9a}$$

Constraint (1a) specifies that each task t in T is executed only on a single resource. This is accomplished by summing all the x_{tri} variables for each task t, and ensuring that the sum is equal to one. Guaranteeing that the assigned start time of all the map tasks is after the job's earliest start time (s_j) is captured by constraint (2a). Constraint (2a) requires iterating through all the x_{tri} variables, specifically focusing on the variables that represent map tasks of the jobs (T_j^{mp}). Furthermore, only the variables where x_{tri}=1 are of

interest because these are the variables that define the assigned start time i of task t on resource r. Recall that constraint (1a) ensures that each task t has only one x_{tri} variable equal to one. Thus, the term $(i|\, x_{tri} = 1)$ identifies the scheduled start time of task t, which is at time i.

Constraint (3a) ensures that the reduce tasks are scheduled to start only after all map tasks are completed. This is accomplished by iterating through all reduce tasks of a job j (T_j^{rd}), and ensuring that the start time of the reduce task is after the completion time of the latest finishing map task (LFMT) of job j. To calculate the completion time of the LFMT, the equation $\max_{t' \in T_j^{mp}, r' \in R, i' \in I} ((i'|x_{t'r'i'} = 1) + e_{t'})$ is used. This equation iterates through all map tasks and calculates the completion time of the task: sum of start time $(i'|x_{t'r'i'} = 1)$ and the execution time $(e_{t'})$. The *max* function returns the maximum value from a given set of values. Constraint (4a) states that N_j, which is initially set to zero, should be changed to one if job j misses its deadline. A job j misses its deadline if the completion time of the latest finishing reduce task (LFRT) in job j is after the job's deadline (d_j). To ensure that N_j is set to one if j misses its deadline, the left-hand side (LHS) is the product of N_j and d_j, and this value must not be less than the right-hand side (RHS), which is equal to the completion time of the LFRT minus d_j. For example, if job 1 has d_1=30s, and the LFRT is 35s, which means job 1 missed its deadline, the RHS is equal to 5s, and the LHS evaluates to 0 since N_j is initially set to zero. To ensure that the LHS is greater than or equal to the RHS, N_j will have to be changed to one, such that the LHS=30, which is greater than the RHS=5.

Making sure that the map and reduce task capacities of each resource are not violated at any point in time is captured by constraints (5a) and (6a), respectively. Constraints (5a) and (6a) use an integer set I^*_{tri} that is defined to contain the assigned start time of task t, if and only if, at time point i, t is still executing on resource r. This set I^*_{tri} is used to ensure that only tasks still executing at a point in 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 (c_r). As shown in Table 1, I^*_{tri} is a set of integers defined as follows: $\{i'|i - e_t < i' \leq i\}$ where i' represent the values in the set I^*_{tri}. The following example task is used to explain the use of I^*_{tri} set. A task, denoted $t1$, has an execution time e_{t1}=5s, and the decision variable x_{tri}=1 (task t is assigned to start executing on resource r at time i) has the following values for its indices: t=$t1$, r=$r1$, i=23, and thus, $x_{t1,r1,23} = 1$. Given the values for $t1$ described, and the current time of interest is i=25s, the set $I^*_{tri} = I^*_{t1,r1,25}$ will have the following values {21, 22, 23, 24, 25}. As shown, this set does contains the assigned start time of $t1$, i=23s. Lastly, constraints (7a) to (9a) specify the valid domain of the decision variables, which restrict the values that the respective variables can have.

3.2 Formulation of the CP Model

The formulation of the CP model is presented in Table 2. Similar to the MILP model, the input of the CP model comprises a set of resources R on which to execute a set of jobs J. Recall, also that a set T contains all tasks of all the jobs in J. The CP model has the following decision variables:

- A binary variable, x_{tr}, which is set to one if task t is assigned to resource r; otherwise, $x_{tr} = 0$ (used for *matchmaking*). There is an x_{tr} variable for each combination of tasks in T, and resources in R.
- An integer variable, a_t, specifies the assigned (or scheduled) start time of a task t (used for *scheduling*). There is an a_t variable for each task in T.
- A binary variable, N_j, which is set to one if job j misses its deadline; otherwise, N_j is set to zero. An N_j variable is defined for each job in J.

The CP constrains are expressed differently than the MILP constraints, but perform the same role as the constraints for the MILP model. The reason for the differences is because the CP model defines a separate decision variable for the assigned start time of the tasks (a_t), as well as makes use of CP's global constraints, and native support for mathematical operators. Constraint (1b) iterates through all tasks in T and ensures that each task is mapped to only one resource. Similar to constraint (1a), this is done by summing all the x_{tr} variables of a given task t, and ensuring the sum is equal to one. Constraint (2b) specifies that the scheduled start time of each map task in a job j (a_t) is at or after job j's start time (s_j). Constraint (3b) states that the scheduled start time of each reduce task of a job j (denoted $a_{t'}$) is at or after the completion time of the LFMT, which is calculated using the *max* function in a similar manner as explained for constraint (3a).

Table 2: Formulation of the CP Model

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

$$\sum_{r \in R} x_{tr} = 1 \qquad \forall\, t \in T \tag{1b}$$

$$\left(a_t \geq s_j \qquad \forall t \in T_j^{mp}\right) \qquad \forall\, j \in J \tag{2b}$$

$$\left(a_{t'} \geq \max_{t \in T_j^{mp}} (a_t + e_t) \qquad \forall t' \in T_j^{rd}\right) \ \forall j \in J \tag{3b}$$

$$\left(\max_{t \in T_j^{rd}} (a_t + e_t) > d_j \implies N_j = 1\right) \qquad \forall j \in J \tag{4b}$$

$$\left(cumulative((a_t|x_{tr} = 1), (e_t|x_{tr} = 1), (q_t|x_{tr} = 1), c_r^{mp}) \ \forall t \in T_j^{mp}\right) \forall r \in R \tag{5b}$$

$$\left(cumulative((a_t|x_{tr} = 1), (e_t|x_{tr} = 1), (q_t|x_{tr} = 1), c_r^{rd}) \ \forall t \in T_j^{rd}\right) \forall r \in R \tag{6b}$$

$$(x_{tr} \in \{0, 1\} \qquad \forall\, t \in T) \ \ \forall r \in R \tag{7b}$$

$$N_j \in \{0, 1\} \qquad \forall\, j \in J \tag{8b}$$

$$a_t \in \mathbb{Z} \qquad \forall\, t \in T \tag{9b}$$

The CP model simplifies the expression of constraint (4a), which ensures that N_j should be changed to one (from zero) if job j misses its deadline, by using the 'implies' operator (see constraint (4b)). A job j misses its deadline if the completion time of the LFRT exceeds the deadline of the job (d_j). The completion time of the LFRT is calculated in a similar manner as in the case of constraint (4a). In addition, constraints (5b) and (6b), which enforce that the map and reduce task capacities of the resources are not violated, are simplified by formulating the constraints using the CP global constraint, *cumulative* [20]. 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: the assigned start time, execution time, and resource requirement of the tasks, as well as the capacity of the resource. There is one cumulative constraint for each resource, and only the tasks that are assigned to that resource (i.e. x_{tr}=1) are of interest for that particular constraint. The remaining constraints, (7b) to (9b), define the domain of the decision variables used in the formulation.

Overall, it can be seen that the constraints in the CP model are expressed in a more intuitive and simple manner. For example, in the formulation of the CP model, constraint (4b) simply uses the logical operator, implies ($\Longrightarrow$) to set N_j to 1 if job j misses its deadline. Furthermore, to formulate constraint (5b) and (6b), the CP model uses CP's global constraint, *cumulative*. Conversely, as

shown in Table 1, the formulation of constraints (4a), (5a), and (6a) for the MILP model requires using more complex mathematical formulas that are not as straightforward.

4. DESIGN AND IMPLEMENTATION EXPERIENCE

Three approaches are used to implement the MILP and CP models presented in Section 3. For all three approaches, after solving the respective MILP or CP model, an output schedule that shows the mapping of tasks to resources, and the scheduled start time of the tasks, is generated. In other words, values are assigned to all the decision variables such that the constraints are *satisfied*, and the objective function is *optimized*. The use of MILP [11] and CP [13] in our resource management techniques led to an optimal solution. Thus, the output schedule that is produced is optimal with regards to the number of jobs that miss their deadlines. This means that there is no other output schedule that can produce a lower number of jobs missing their deadlines.

4.1 Approach 1: MILP Model with LINGO

LINGO is a tool used to build, model, and solve optimization problems (through mathematical programs) developed by LINDO Systems Inc. [23]. 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.

This section briefly discusses how the MILP model was implemented in LINGO v13.0. More detail on how to use LINGO can be found in [23]. The LINGO modeling language provides a data type called *Sets* that can be used to model a group of related objects. By using Sets, constraints on the decision variables can be efficiently and compactly expressed using a single statement. Each set can have a number of attributes associated with each member of the set. In the implementation of the MILP model, sets were used to represent the jobs set *J*, tasks set *T*, resources set *R*, and time range set *I*. For example, the task set *T* 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 execution time and resource requirement attributes represent e_t and q_t, respectively.

A representative set of examples of how the constraints of the MILP model (defined in Table 1) are implemented using LINGO are presented. Constraint (1a) 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 members in the given set. The variable *x* used in the LINGO model has the same role as the *x* decision variable discussed in Section 3.1.

The implementation of Constraint (5a) 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))
);
```

The @SUM construct uses 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 will be processed. For example, the first @SUM construct specifies that only tasks with a *type* attribute equal to zero (i.e. map tasks) are processed.

An important feature in this implementation is captured in how constraint (4a) 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. The *If-Then-Else* construct could have been used to simplify the implementation of constraint (4a) 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 (4a) changed the program from a MILP into a Mixed Integer Non-linear Program (MINLP). MINLPs are generally more difficult and require more processing time to solve compared to MILPs [23], and this leads to a longer time before a solution can be found. Thus, the use of the *If-Then-Else* construct was avoided.

4.2 Approach 2: CP Model with MiniZinc and Gecode

In Approach 2, the CP model is implemented with MiniZinc 1.6 [15], which is an open-source CP-based 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 [15] 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) [16]. Gecode is an open-source tool implemented in C++ for solving CP problems.

This section briefly discusses how the CP model was implemented using MiniZinc. More detail on how to use the MiniZinc modeling language can be found in [24]. Similar to LINGO, MiniZinc also provides a mechanism to group together closely related data called *Sets and Arrays*. In MiniZinc, the data set for tasks is implemented as follows:

```
set of int: Jobs = 1..NUM_JOBS;
set of int: Tasks = 1..NUM_TASKS;
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 to represent the indices of the arrays. Next, the attributes of the tasks, which are the same as those discussed in Section 4.1, are declared using arrays. The domain of each of the attributes, which is the range of acceptable values that an attribute can have, is also declared here. For example, the domain of the parent job attribute is equal to the set of integers called Jobs, which has a range from 1 to *NUM_JOBS* where *NUM_JOBS* is the number of jobs in the batch that needs to be executed. As shown, 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 performs the same function.

A representative set of examples of how the CP constraints (defined in Table 2) are implemented using MiniZinc is presented. In MiniZinc, constraint (2b) 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 constraints in MiniZinc, start with the keyword *constraint*. The *forall* construct performs an identical function to LINGO's @FOR construct. Similarly, the *where* keyword in the forall statement is MiniZinc's conditional qualifier operator. The /\ operator performs a logical conjunction (logical *and*) operation.

A novelty of this implementation is the devising of a modified cumulative constraint for implementing constraints (5b) and (6b). The original cumulative constraint provided by MiniZinc [15] could not be used because it was not able to handle the two different task types present in MapReduce jobs: map tasks and reduce tasks. Thus, a modified cumulative constraint, called *mycumulative*, is developed that ensures that map tasks and reduce tasks are only scheduled on the map slots and reduce slots of the resources, respectively, and also the capacities of the resources are not violated. The function prototype for the *mycumulative* constraint is presented:

```
predicate mycumulative(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: start time of the tasks, execution time of the tasks, resource requirement of the tasks, and the capacity of the resources, are the parameters in the original cumulative function provided by MiniZinc. The new parameters added include: the matchmaking variable *x* (discussed in Section 3.2), the *type* attribute of the tasks, and a variable *taskType* which indicates if the constraint should be computed for map tasks (taskType=0), or for reduce tasks (taskType=1). Another change made in *mycumulative* is that it ensures that the resource capacities are not violated for all the resources in *R,* within the function, which means that the *mycumulative* constraint needs to be invoked only once. The cumulative constraint provided by MiniZinc checks only a single resource within the function, and thus needs to be invoked separately for each resource.

A code snippet of the *mycumulative* 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 range of times in the *Times* set is calculated 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. The *bool2int* library function converts a Boolean value to an integer, where true is equal to one, and false is equal to zero. The bool2int component of the equation is used to ensure that only tasks that are still executing at the time of interest, *i*, are included in the resource capacity calculations.

4.3 Approach 3: CP Model with CPLEX

In Approach 3, the CP model is implemented and solved using IBM CPLEX 12.5 [17]. More specifically, CPLEX's *Optimization Programming Language* (OPL) [25] is used to implement the CP model. OPL is an algebraic language specifically designed for expressing optimization problems, and therefore is able to provide a natural representation of optimization models that is more compact and less complex than using general-purpose programming languages. The OPL model is then 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 [26][27]. 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 executed. The interval data type has five attributes: start time, duration, end time, optionality, and intensity. The optionality attribute is used to indicate whether or not the interval is required to be present in the solution. For example, the optionality attribute can be used to represent 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. The intensity attribute defines the resource usage or utility of a task over its interval.

The implementation of the CP model using CPLEX is briefly discussed. Additional information for expressing CP models in OPL can be found in [25] and [26]. Similar to the other approaches, OPL supports using sets and a data type called *tuple* which allows related data to be grouped together. 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, and then this tuple is used to define a set of *Tasks*. The task tuple has the same attributes as those discussed for Approaches 1 and 2, except for an additional field called *id* which is required in OPL to uniquely identify the task.

A key feature of this implementation is that it makes use of CPLEX's *tuple* sets and *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 [26]. This in turn allows the system to efficiently solve the matchmaking and scheduling problem by reducing processing time and memory requirements [27]. More specifically, the CP model's decision variables: a_t and x_{tr} are implemented using CPLEX's *interval* data type, and are named *taskInterval* and *xtr*, respectively:

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

There is a *taskInterval* variable for each task that needs to be mapped and scheduled, and this interval defines the task's start time, end time, and execution time. There is also an x_{tr} variable for each tuple in the Options set, which is a derived set that contains all the possible combinations of tuples of the form <Task, Resource>. Note that this interval is *optional*, which allows only a subset of the intervals to be present in the final schedule. By using the interval data type, the implementation can make use of the optimized library functions that CPLEX provides.

A representative set of examples of the implementation of the constraints of the CP model (defined in Table 2) is presented. For instance, in the OPL model, constraint (1b) is expressed using the *alternative* constraint as follows:

```
forall (t in Tasks)
   alternative(taskInterval[t], all(o in Options:
               o.task.id==t.id) xtr[o]);
```

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 they both start and end at the same time. Thus, it is appropriate to use the alternative constraint to express constraint (1b), which ensures that each task is assigned to only one resource. In the example, the set *S* is produced by using the *all* construct invoked with a conditional qualifier (':' operator). More specifically, *S* is a subset of x_{tr} variables that have the same id as the task of interest, *t*.

In the OPL model, constraint (5b) 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 is used to generate the resource usage of a task, and requires two parameters: an interval *i* to represent the task, and a height value *h* to indicate the resource usage (i.e. capacity requirement) required by the task. The pulse function produces a value as a function of time. When the task is active (i.e. during the interval between the start and end times), the pulse function generates a value equal to the supplied value *h* to indicate the amount of resource usage of the task, and at all other points in time, the pulse function generates a value of zero. The expression for constraint (5b) states that for each resource *r*, the sum of all the values produced by the pulse function at each point in time, must be less than or equal to the map capacity of resource *r*.

5. PERFORMANCE EVALUATION

To evaluate the system performance achieved with the three approaches discussed in Section 4, experiments were performed on a closed system using various batch workloads where each batch comprised of multiple jobs to execute. Each experiment concluded after successfully mapping and scheduling all the jobs in the batch, and an output schedule and completion time of the batch is determined. Such an experimental environment, based on a closed system is similar to what is used by [8], [9], and [10], and is apt for evaluating and comparing the performance (e.g. processing time) of the modeling techniques and solvers. In future work, we will investigate techniques to handle an open system with a stream of job arrivals.

To compare the performance of the three approaches the following metrics are used:

- *Completion time* (*C*): time at which all jobs in the batch finish executing.
- *Processing time* (*P*): time it takes for the solver to read the input data (job, task, and resource sets), generate the model, and produce the output schedule that minimizes the number of late jobs.
- Number of jobs that miss their deadlines (*N*).
- Size of workload (number of tasks) the approach could successfully handle.

Note that the system focuses on meeting deadlines of the jobs in the workload and its 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 experiments were conducted on a PC with a 3.2GHz Intel Core 2 Duo CPU and 6.00GB of RAM running under Windows 7 Professional. Lower processing times for obtaining the solutions can be expected to be achieved by running the solvers on a system with a faster CPU and more memory. Each experiment was repeated ten times and the confidence intervals, which were all less than 8% at a confidence level of 95%, are shown on the figures as bars originating from the mean value.

5.1 Description of Workloads

Table 3 presents the system and workload parameters for the experiments used to compare the three approaches. The workloads are synthetic workloads, each of which is characterized by a number of parameters. Similar workloads have been used by other researchers. For example, the Large 2 workload is adopted from [10], whereas the other workloads are derived by using the same distributions as those used in [10].

A walkthrough of Table 3 is provided. In the 'Jobs' column, the first row defines the number of jobs in the batch (*n*). The second and third rows define the earliest start time (s_j) and deadline (d_j) of each job *j*, respectively. The last row(s) of the Job's column denotes the number of map tasks (k_j^{mp}) and reduce tasks (k_j^{rd}), respectively, for job *j*. The next column, 'Task Execution Times', specifies the execution times of map tasks (e_t^{mp}) and reduce tasks (e_t^{rd}), respectively. 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 slots (c_r^{mp}) and reduces slots (c_r^{rd}) are defined. Since the workload and system parameters are integers, discrete uniform distributions (DU) are used to generate the values for all parameters except d_j. The calculation of d_j uses a uniform distribution (U), which produces real values, for generating a *multiplier* for e_j^{max}—the execution time (in seconds) of job *j* when all tasks are executed sequentially (i.e. max execution time of job *j*). To ensure that d_j is an integer, the *ceiling* function is used at the end of the calculation. Note that in the 'Large 2' row, $e_j^{tot\_mp}$ (in seconds) denotes the total execution time of all map tasks of job *j*.

Table 3: System and Workload Parameters

Workload	Jobs, *J* (s_j and d_j in seconds, s)	Task Execution Times (in seconds, s)	Resources, *R*
Small 1	n =5: $s_j \sim$ DU(1,50) $d_j \sim s_{j+}(e_j^{max})*$ U(1,5) k_j^{mp}=10, k_j^{rd}=3	$e_t^{mp} \sim$ DU(1,15) $e_t^{rd} \sim$ DU(1,50)	m =10: c_r^{mp}=2 c_r^{rd}=2
Small 2	n =5: $s_j \sim$ DU(1,50) $d_j \sim \lceil s_j + e_j^{max} * \mathrm{U}(1,2) \rceil$ $k_j^{mp} \sim$ DU(1,15) $k_j^{rd} \sim$DU(1, k_j^{mp})	$e_t^{mp} \sim$ DU(1,15) $e_t^{rd} \sim$ DU(1,75)	m =25: c_r^{mp}=2 c_r^{rd}=2
Medium	n =10: $s_j \sim$ DU(1,50) $d_j \sim \lceil s_j + e_j^{max} * \mathrm{U}(1,2) \rceil$ k_j^{mp}=10 k_j^{rd}=5	$e_t^{mp} \sim$ DU(1,25) $e_t^{rd} \sim$ DU(1,75)	m =15: c_r^{mp}=2 c_r^{rd}=2
Large 1	n =2: s_1= 0, s_2= 500 $d_j \sim \lceil s_j + e_j^{max} * \mathrm{U}(1,2) \rceil$ k_j^{mp}=100 k_j^{rd}=30	$e_t^{mp} \sim$ DU(1,15) $e_t^{rd} \sim$ DU(1,50)	m =25: c_r^{mp}=4 c_r^{rd}=4
Large 2 (adopted from [10])	n =50: $s_j \sim$ DU(1,1500) $d_j \sim \lceil s_j + e_j^{max} * \mathrm{U}(1,2) \rceil$ $k_j^{mp} \sim$DU(1,100) $k_j^{rd} \sim$DU(1, k_j^{mp})	$e_t^{mp} \sim$ DU(1,10) $e_t^{rd} = \frac{e_j^{tot\_mp}}{k_j^{rd}}$	m =50: c_r^{mp}=2 c_r^{rd}=2

The goal of the experiments is to use various workloads with different characteristics such as the size of the batch, the number of tasks in a job, and the execution times of tasks, for analyzing the impact of workload characteristics on performance. For example, in the Small 1 workload there are five jobs, each job with 10 map tasks with execution times varying from 1s to 15s, and three reduce tasks with execution times varying from 1s to 50s. 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 compared to the Small 1 workload, which has 65 tasks.

5.2 Results of Experiments

5.2.1 Small and Medium Workloads

Figure 2 and Figure 3 present the *C* and *P* results, respectively, for the three approaches when using the small and medium workloads. In all the experiments performed, the optimal solution is found in the sense that *N* is zero. As expected, the results show that for all three approaches: as the size of workload increases giving rise to a larger number of tasks, *P* and *C* also increase. From Figure 3, it can be observed that Approach 3 achieves the lowest *P* (note that the bars are quite small and may not be visible); however, it also generated an output schedule that produced the highest *C*. This can be attributed to the fact that in Approach 3 the solver produces the first output schedule that optimizes the objective function (minimizing *N*) and does not focus on the minimization of *C*. The lower *P* achieved by Approach 3 can be 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 [26] (as discussed in Section 4.3).

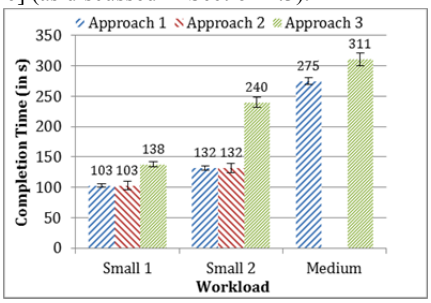

Figure 2. Completion time for the small & medium workloads.

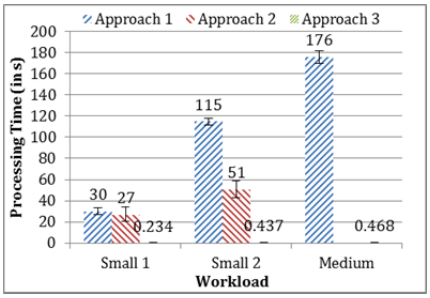

Figure 3. Processing time for the small & medium workloads.

Another observation that can be made from Figure 3 is that the approaches that implement the CP model (i.e. Approaches 2 and 3) attained a lower *P* compared to Approach 1, which implements the MILP model. The reason for this behavior can be due 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 that there is an x_{tri} variable for each combination of tasks in *T*, resources in *R*, and time points in *I*. In the CP model there are less decision variables because there are separate decision variables for matchmaking, x_{tr}, and scheduling, a_t. Note that Approach 2 was not able to handle the Medium workload after a couple of hours of solving. This may be due to the limitations of the solver from being able to handle such a large number of tasks to map and schedule on our system, which leads to a model that contains a large number of decision variables.

5.2.2 Large Workloads

The *C* and *P* results for the three approaches when handling the large workloads are shown in Figure 4. Approach 2 was not able to handle these larger workloads for the same reasons as discussed in Section 5.2.1, and Approach 1 was only able to generate an output schedule for the Large 1 workload. When attempting to generate solutions for the larger workloads with Approaches 1 and 2, the system would eventually run out of memory, and the solver would crash. The solvers of Approach 1 and 2 could not handle such a large number of decision variables on our system. The results show that Approach 3 outperforms Approach 1 for similar reasons as discussed in Section 5.2.1.

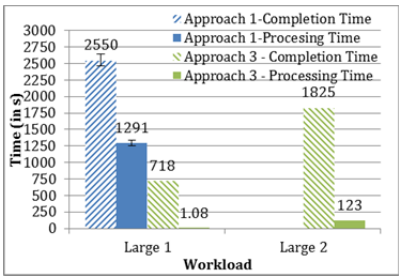

Figure 4. Completion time and processing time for the large workloads.

In order, for Approach 1 to handle the Large 1 workload, the granularity of *I* was reduced to decrease the number of decision variables in the model. Recall from Section 3.1 that Approach 1 requires specifying a set of integers, *I*, which defines the range of time (or time slots) during which jobs can be scheduled to start executing on a resource. The time range can be chosen from time i=0 to i=*MAX_COMP_TIME* where *MAX_COMP_TIME* is the maximum completion time of the workload given that each job executes sequentially on the *m* resources. The granularity of *I* can be changed to restrict when jobs can be executed. For example, the granularity of a set I_1={1, 2, 3, …, 100} can be reduced to I_2={2, 4, 6, …, 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 the solver to generate and solve the MILP model used in Approach 1 because more decision variables are present. The MILP model has a decision variable, x_{tri}, for each combination of tasks *t* in *T*, resources *r* in *R*, and time *i* in *I*. As such, the number of variables that are present in MILP model increases as the number of tasks, number of resources, or number of time slots increase.

For the experiment where Large 1 was being used, the set *I* for Approach 1 was set to have 100 time slots with an interval of 25 seconds between each slot: {0, 25, …, 2500}. If reducing the granularity of *I* was not done, 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 an output schedule. A disadvantage of reducing the granularity of *I* is that this procedure can increase *C* because some tasks cannot be scheduled to start executing at their earliest start times. For example, if a job *j* has s_j =27s, and the time slots have intervals of 25s, the tasks of *j* cannot be executed until time 50s. Figure 4 shows *C* for Approach 1 is over 2500s, which is about three times longer than Approach 3's *C*. Therefore, the results show that for Approach 1, there is tradeoff between being able to handle larger workloads, and achieving a lower *C*.

5.2.3 Effect of Workload Parameters

In this section, the effect of varying different workload parameters on system performance is discussed. The experiments conducted in this section are based on the Large 2 workload (adopted from [10]). Approach 3 was the only approach capable of handling the larger workloads with up to 100 jobs and 7000 tasks that were experimented with. As discussed in Section 5.2.2,

Approaches 1 and 2 could not handle larger workloads because the system would crash due to lack of memory. Approach 3's use of CPLEX's CP Optimizer solving engine provides mechanisms and functions to efficiently solve scheduling problems [26]. As discussed in Section 4.3, implementing the CP model using CPLEX's interval decision variables allows the solver to efficiently use the system memory, which in turn allows larger workloads to be handled by Approach 3. Note that for all the experiments discussed in this section N was zero.

***Effect of number of jobs* (*n*):** Figure 5 shows C and P when n is varied for the Large 2 workload. As expected, C increases with n because there are more jobs to execute. In addition, P also increases because there are more tasks to map and schedule on the resources. It is observed P/C, a measure of resource management overhead, increases with n. This shows that P is increasing at a faster rate than C. For example, the results show that the highest P/C is 0.26 (26%), and is achieved when n=100. In some systems, this scheduling overhead may be too high; however, the overhead can be tolerated in situations where the task to resource mapping and scheduling for the batch of jobs is performed offline and the execution of the batch takes place at a later point in time. When n is less than 100, P/C is much smaller (0.0196 and 0.06074 for n = 25 and 50, respectively), and thus, in these situations, online mapping and scheduling can be considered: the solver can be run as soon as the batch of jobs becomes available on the system followed by the execution of the batch.

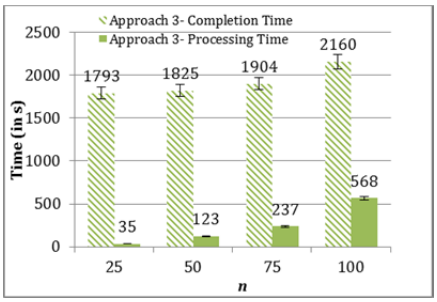

Figure 5. The effect of number of jobs on performance.

***Effect of task execution time*:** Figure 6 shows P and C when the upper-bound of the discrete uniform distribution used for generating task execution times, denoted , is varied for the Large 2 workload. As shown in Figure 6, P increases with because there is now a higher chance of tasks having overlapping execution. Thus, the solver requires more time to decide at what time and on which resource to execute a task in order to generate an output schedule that minimizes N. As expected, C also increases because jobs require more time to execute. However, it can be observed that P increases at a slower rate compared to C, and thus, the P/C decreases as increases. The resource management overhead is observed to be small: P/C varies from 0.0674 to 0.0320 as is changed from 10 to 100 seconds.

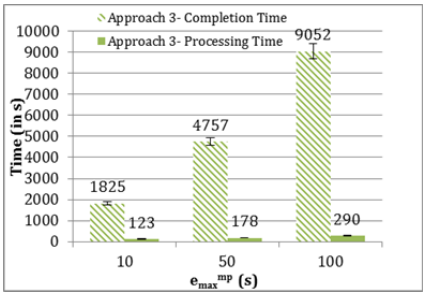

Figure 6. The effect of task execution time on performance.

A number of experiments were performed to analyze the effect of changing the other workload and system parameters, including: s_j, d_j, and m on system performance. Due to space limitations, only a representative set of results is presented. The following modified versions of the Large 2 workload were used: (1) *Large 2a*: same as Large 2, but increases s_j to ~DU(1,*3000*); (2) *Large 2b*: same as Large 2, but increases d_j to ~ s_{j+} (e_{max})* U(1,*4*); and (3) *Large 2c-1*, *2c-2*, and *2c-3:* same as Large 2, but sets m to 10, 25, and 100, respectively. Figure 7 shows the results for these additional Large 2 workloads.

***Effect of earliest start time* (s_j):** Figure 7 shows that increasing s_j (see Large 2 and Large 2a) increases C because on an average, jobs tend to start executing at a later time. Conversely, increasing s_j reduces P. Having a larger range of s_j decreases the chance of jobs having overlapping execution times, and also reduces the contention for resources. This means that at a given point in time, there may not be as many concurrent tasks that the solver has to map and schedule compared to the situation where the s_js are closer to one another. Thus, the solver is able to quickly determine an output schedule that ensures that N is minimized.

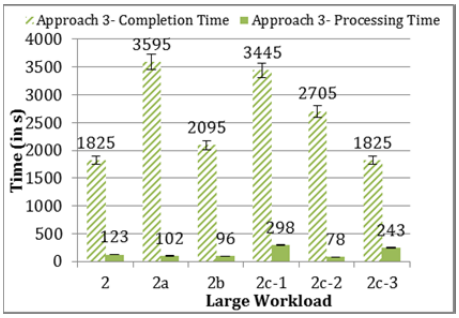

Figure 7. System performance for the additional Large 2 workloads.

Effect of deadline (d_j): When comparing the Large 2 and Large 2b workloads of Figure 7, it is observed that increasing d_j reduces P, but increases C. When the deadlines of the jobs are not as stringent, jobs will have more *slack time* (also called laxity), which is defined as the difference between the deadline, and the sum of the execution time and the earliest start time of the job: d_j – (s_j +). The slack time is the extra time a job has to complete its execution before its deadline. When the slack time is higher, the solver does not need to spend as much time to generate an output schedule that minimizes N. The increase in C can be attributed to the fact that the solver returns the first output schedule that is able to minimize N, and does not focus on minimizing C. When the jobs have smaller slack times, the solver has to ensure that jobs are completed in shorter periods of time, which in turn reduces C.

***Effect of number of resources* (*m*):** Figure 7 shows that increasing m from 10 to 25 (see Large 2c-1 and Large 2c-2), both P and C decrease because there are more resources in which to map and schedule the tasks. Even though there are less decision variables to generate and solve when m is smaller, the solver requires more time to determine the best task to map and schedule on the resources at a given time so that N is minimized. It is observed that when m is increased from 25 to 50 (Large 2c-2 and Large 2), P increases because the solver has more decision variables to generate and solve. However, there are more resources available to execute the tasks, which leads to a lower C. Lastly, when increasing m from 50 to 100 (Large 2 and Large 2c-3), P increases, whereas C stays the same. In this case, the additional resources cannot be used to further decrease N or C because both N and C are already minimized, and thus increasing

m just increases the number of decision variables that the solver has to generate and solve, which adds unnecessary overhead, and leads to higher *P*. Therefore, it can be observed that for a given workload, changing *m* to a value that is too high or too small can lead to an increase in *P*. In addition, increasing *m* tends to reduce *C* until *m* is significantly high and no further improvement in *C* is observed.

5.2.4 Summary of Experimental Results

This section summarizes the key observations made from analyzing the results of the experiments. Recall that solving an MILP [11] and CP [13] generates optimal solutions, and therefore all three approaches generated optimal output schedules with regards to minimizing *N*. For the system and workload parameters experimented with, and for the workloads that the approaches could handle, optimal output schedules where N=0 were generated. For workloads in which *N* was not be zero, a task mapping and schedule that minimizes *N* is generated.

***Approach 1*:** did 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* for the small workloads; however, for a given workload, Approach 1 was measured to have a higher *P* compared to Approach 2. In addition, for the Medium workload, Approach 1 generated an output schedule with 11.5% lower *C* compared to Approach 3, but *P* was also 375% higher. Lastly, for the Large 1 workload, Approach 1 was outperformed and had higher *C* and *P* compared to Approach 3. Thus, for the system and workload parameters experimented with, it is not recommended that Approach 1 be used unless *P* 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 mapping and scheduling for the jobs can be performed ahead of time (e.g. offline).

***Approach 2*:** is only able to handle the smaller workloads (less than 150 tasks) on our system. For a larger value for the total number of tasks in the batch, Approach 2 could not generate a schedule because the system used would eventually run out of memory, and the solver would crash. As discussed, along with Approach 1, Approach 2 generated an output schedule with the lowest *C* for the small workloads. Even though, *P* was lower compared to Approach 1, Approach 2's *P* is still over 100 times larger than the *P* measured for Approach 3. Thus, for the small workloads, there is a trade-off between having a lower *C* (using Approach 2) versus a lower *P* (using Approach 3). Similar to Approach 1, Approach 2 can be considered for small workloads when the resource management can be performed ahead of the time at which the batch becomes ready to execute.

Approach 3: In general, the experimental results showed that Approach 3 performed the best. Regardless of workload size, it was able to achieve a much lower *P* compared to the two other approaches. However, it also generated an output schedule with slightly higher *C*. For example, for the Small 2 workload, the *C* is 1.8 times larger compared to Approaches 1 and 2, however; the *P* is over 100 times smaller. On many systems satisfying the deadlines is sufficient and achieving a small batch completion time is only a secondary objective. Furthermore, Approach 3 is able to handle the larger workloads (i.e. Large 2 and above) that the other two approaches could not handle. In fact, the experiments described in this paper indicate Approach 3 is able to handle workloads containing up to 7000 tasks (see Figure 5).

Overall, the experimental results indicated that Approach 3 would be the best candidate to implement a resource manager that is capable of handling an open stream of requests arriving on the system that is being considered for our future research. Approach 3 was the only approach capable of handling the larger workloads, and was measured to have the lowest *P*. Having a low *P* is important to consider when handling an open stream of job requests, because a low matchmaking and scheduling overhead is key to efficiently process incoming requests.

6. CONCLUSIONS

This paper concerns resource management on clouds in which the workload includes requests characterized by multiple stages of execution, and an end-to-end SLA. More specifically, our work focuses on engineering resource management middleware that can effectively perform matchmaking and scheduling of MapReduce jobs, each of which is characterized by an end-to-end SLA comprising an earliest start time, execution time, and a (soft) deadline specified by the user. Both the reduction of resource management overhead as well as achieving high system performance are objectives of this research. The problem of matchmaking and scheduling MapReduce jobs with SLAs was formulated using MILP and CP. The MILP and CP models were implemented and solved using three approaches: (1) MILP model implemented and solved using LINGO [6], (2) CP model implemented using MiniZinc/FlatZinc [7] and solved using Gecode [8], and (3) CP model implemented and solved using IBM CPLEX. All three approaches have an associated learning curve period; however, configuring, implementing, and executing the models using Approaches 1 and 3 were easier compared to Approach 2 because both LINGO and CPLEX provide a feature-rich integrated development environment (IDE), whereas MiniZinc and Gecode only provide command-line interfaces.

Solving an MILP or CP model generates optimal solutions, and therefore all three approaches are able to produce optimal output schedules with regards to minimizing the number of jobs missing their deadlines. Our investigation and experiences with using the various techniques and software packages to formulate and solve the matchmaking and scheduling problem were discussed. A number of experiments were performed using different workloads and parameters to compare the performance of the three approaches in terms of metrics such as completion time (*C*): time at which all jobs in the workload finish executing, and processing time of the solver (*P*). Insights into system behaviour gained from the experimental results for the workload and system parameters we experimented with are presented.

- Approach 3 is observed to achieve the lowest *P* compared to the two other approaches; however, it also generated an output schedule that produced the highest *C*. In addition, Approach 3 was the only approach able to handle the larger workloads (over 1000 tasks in the workload, described in Section 5.2.3).
- Approaches 1 and 2 each had a case where they were able to generate an output schedule that had the lowest *C*; however, the *P* in these cases is much higher compared to Approach 3.
- The results show that Approaches 2 and 3, which use CP, have lower *P* compared to Approach 1, which uses MILP.
- Approach 3 was observed to effectively handle the Large 2 workload that was adopted from [10]: the processing time was only 6.7% of the batch completion time.

A more detailed analysis of Approach 3 that includes the effect of larger workloads was performed and the insights gained are discussed.

- Both *P* and *C* are observed to increase when the number of jobs (n) in the workload increases. The *P* to *C* ratio, denoted *P* /*C*, which is an indicator for the mapping and scheduling overhead, also increases with n. However, when n <=75 it was found that *P* /*C* was reasonably small: less than 0.13.

- When execution time of tasks increases, both P and C increase as well; however, in this case, P/C decreases as execution time of tasks increase.
- For a given workload, if the number of resources in the system (m) is too small or too high, P tends to increase, but having a higher m typically can generate an output schedule with lower C until a point where C can no longer be decreased. Increasing m when the workload does not require it (i.e. number of tasks is not sufficiently large), tends to increase P.
- Increasing the deadline of the jobs (d_j) reduces P, but increases C. Similarly, increasing the earliest start times of the jobs (s_j) decreases P, and tends to increase C as well.

If minimizing the number of jobs missing their deadlines is the sole objective, Approach 3 that is able to handle workloads with over 1000 tasks seems to be the most suitable because of the lower P. However, it was observed that using Approach 3 leads to a slightly higher C in comparison to the other approaches. Based on the results of the experiments described in this paper, it was found that Approaches 1 and 2 are most useful in cases where the workloads are smaller (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). Approach 3 would be best suited to implement a resource manager that can perform matchmaking and scheduling of an open stream of MapReduce jobs with end-to-end SLAs. Such a resource manager warrants further investigation. Our plans for future research also includes refining the optimization models to consider more advanced features for resource management of MapReduce jobs including data locality and speculative execution (backup tasks) [5]. Evaluation of the three approaches in the context of more complex and real workloads, and investigating techniques for handling node failures, which are important for large systems, form important directions for future research as well.

7. ACKNOWLEDGMENTS

We are grateful to Huawei Technologies, Canada, and the Government of Ontario for supporting this research.

8. REFERENCES

[1] Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., and Brandic, I. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems.* 25, 6 (June 2009), 599-616.

[2] Heinz, S., and Beck, J.C. 2011. Solving resource allocation/scheduling problems with constraint integer programming. *In Proc. of Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS)* (12-13 June 2011). 23–30.

[3] Hooker, J.N. 2005. Planning and scheduling to minimize tardiness. *In van Beek, P., ed., Principles and Practice of Constraint Programming*. Vol. 3709 of LNCS (2005). 314–327.

[4] Verma, A., Cherkasova, L., Kumar, V.S., and Campbell, R.H. 2012. Deadline-based workload management for MapReduce environments: Pieces of the performance puzzle. In *Proc. of Network Operations and Management Symposium (NOMS)* (16-20 April 2012). 900-905.

[5] Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. *International Symposium on Operating System Design and Implementation* (December 2004). 137–150.

[6] The Apache Software Foundation. Hadoop. Available: http://hadoop.apache.org.

[7] Apache. Hadoop Wiki. Available: http://wiki.apache.org/hadoop/PoweredBy

[8] Kc, K., and Anyanwu, K. 2010. Scheduling Hadoop Jobs to Meet Deadlines. In *Proc. of International Conference on Cloud Computing Technology and Science (CloudCom)* (Nov. 30 2010-Dec. 3 2010). 388-392.

[9] Dong, X., Wang, Y., and Liao, H. 2011. Scheduling Mixed Real-Time and Non-real-Time Applications in MapReduce Environment. In *Proc. of International Conference on Parallel and Distributed Systems (ICPADS)* (7-9 Dec. 2011). .9-16.

[10] Chang, H., Kodialam, M., Kompella, R.R., Lakshman, T.V. Lee, M., and Mukherjee, S. 2011. Scheduling in mapreduce-like systems for fast completion time. In *Proc. of IEEE INFOCOM* (10-15 April 2011). 3074-3082.

[11] Bosch, R. and Trick, M. 2005. Integer programming. *Search Methodologies*. Springer US (2005). 69-95.

[12] Chinneck, J.W. 2004. Chapter 13: Binary and Mixed-Integer Linear Programming. *Practical Optimization: a Gentle Introduction* (2004). Available: http://www.sce.carleton.ca/faculty/chinneck/ po.html

[13] Rossi, F., Beek, P., and Walsh, T. 2008. Chapter 4: Constraint Programming. *Handbook of Knowledge Representation* (2008). 181-211.

[14] Lindo Systems Inc. Lindo Systems – Optimization Software. Available: http://www.lindo.com/.

[15] NICTA. MiniZinc and FlatZinc. Available: http://www.MiniZinc.org/.

[16] Gecode. Generic Constraint Development Environment. Available: http://www.gecode.org/.

[17] IBM. IBM ILOG CPLEX Optimization Studio. Available: http://www-03.ibm.com/software/products/us/en/ibmilogcpleoptistud

[18] Lustig, I. J., and Puget, J.-F. 2001. Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. *INTERFACES*. 31, 6 (Nov.-Dec. 2001). 29-53.

[19] Refalo, P. 2000. Linear formulation of constraint programming models and hybrid solvers. *Principles and Practice of Constraint Programming–CP 2000*. Springer Berlin Heidelberg (2000). 369-383.

[20] Beldiceanu, N. and Demassey, S. Global Constraint Catalog. Available: http://www.emn.fr/z-info/sdemasse/gccatold/Ccumulative.html.

[21] Udupi, Y. and Dutta, D. Business Rules and Policies driven Constraints-based Smart Resource Placement in Openstack. White Paper. Cisco.

[22] Van den Akker, J. M., Hurkens, C., and Savelsbergh, M. 2000. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*. 12.2 (2000). 111-124.

[23] LINDO Systems Inc. 2011. LINGO 13.0: User's Guide.

[24] Marriott, K., Stuckey, P.J., Koninck, L.D., and Samulowitz, H. 2012. An Introduction to MiniZinc Version 1.6.

[25] IBM. 2009. IBM ILOG OPL Language Reference Manual. White Paper. IBM Corporation (2009).

[26] IBM. 2010. Detailed Scheduling in IBM ILOG CPLEX Optimization Studio with IBM ILOG CPLEX CP Optimizer. White Paper. IBM Corporation (2010).

[27] Dong, T. 2009. Efficient modeling with the IBM ILOG OPL-CPLEX Development Bundles. White Paper. IBM Corporation (December 2009).