A Responsive Knapsack-based Algorithm for Resource Provisioning and Scheduling of Scientific Workflows in Clouds

Maria A. Rodriguez and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computing and Information Systems, The University of Melbourne, Australia
e-mail: mariaars@student.unimelb.edu.au, rbuyya@unimelb.edu.au

***Abstract*—Scientific workflows are used to process vast amounts of data and to conduct large-scale experiments and simulations. They are time consuming and resource intensive applications that benefit from running in distributed platforms. In particular, scientific workflows can greatly leverage the ease-of-access, affordability, and scalability offered by cloud computing. To achieve this, innovative and efficient ways of orchestrating the workflow tasks and managing the compute resources in a cost-conscious manner need to be developed. We propose an adaptive, resource provisioning and scheduling algorithm for scientific workflows deployed in Infrastructure as a Service clouds. Our algorithm was designed to address challenges specific to clouds such as the pay-as-you-go model, the performance variation of resources and the on-demand access to unlimited, heterogeneous virtual machines. It is capable of responding to the dynamics of the cloud infrastructure and is successful in generating efficient solutions that meet a user-defined deadline and minimise the overall cost of the used infrastructure. Our simulation experiments demonstrate that it performs better than other state-of-the-art algorithms.**

I. Introduction

Workflows are defined as a set of computational tasks and a set of data or control dependencies between them. They are widely used by the scientific community to analyse and process large amounts of data efficiently. These large-scale scientific workflows are resource-intensive applications and hence are commonly deployed on distributed platforms. Scheduling algorithms play a crucial role in running workflows efficiently as they are responsible for the orchestration of the tasks on the distributed compute resources. Their decisions are guided by a collection of Quality of Service (QoS) requirements defined by the application users such as minimising the total cost or makespan (i.e. total execution time), or meeting a specified budget or deadline. This scheduling problem is non-trivial, in fact, it is a well-known NP-complete problem [1] and therefore, algorithms must focus on finding approximate solutions in a reasonable amount of time.

Infrastructure as a Service (IaaS) clouds offer an easily accessible, flexible, and scalable infrastructure for the deployment of large-scale scientific workflows. They allow users to access a shared compute infrastructure on-demand while paying only for what they use. This is done by leasing virtualised compute resources, or Virtual Machines (VMs), with a predefined CPU, memory, storage and bandwidth capacity. Different resource bundles (i.e. VM types) are available to users at varying prices to suit a wide range of application needs. Aside from VMs, IaaS providers also offer storage services and network infrastructure to transport data in, out, and within their facilities. To fully take advantage of these services and opportunities, scheduling algorithms must consider several key characteristics of clouds.

The first one is the on-demand, elastic resource model. This feature suggests a reformulation of the scheduling problem as traditionally defined for other distributed platforms such as grids and clusters. Clouds do not offer a finite set of compute resources, instead, they offer a virtually infinite pool of VMs with various configurations ready to be leased and used only for as long as they are needed. This model creates the need for a resource provisioning strategy that works together with the scheduling algorithm; a heuristic that decides not only the type and number of VMs to request from the cloud but also when is the best time to lease and release them. Since this work is tailored for cloud environments, from here on, the word *scheduling* will be used to refer to an algorithm capable of making both, resource provisioning and scheduling decisions.

Another feature to consider is the utility-based pricing model used by cloud providers. The cost of using the infrastructure needs to be considered or otherwise, users risk paying prohibitive and unnecessary costs. For example, the number of VMs leased, their type, and the amount of time they are used for, all have an impact on the total cost of running the workflow in the cloud. Consequently, schedulers need to find a trade-off between cost and other QoS requirements such as makespan.

A third characteristic of clouds is their dynamic state and the uncertainties this brings with it. An example is the variability in performance exhibited by VMs in terms of execution times [2]. This variability means that despite a VM type being advertised to have a specific CPU capacity, it will most likely perform at a lower capacity that will change overtime. It also means that two VMs of the same type may exhibit completely different performances. Furthermore, having multiple concurrent users sharing a network means that performance variation is also observed in net-

working resources [2]. Yet another source of uncertainty are the VM provisioning and deprovisioning delays [3]; there are no guarantees on their values and they can be highly variable and unpredictable. Recognising performance variability is important for schedulers so that they can recover from unexpected delays and fulfil the QoS requirements.

As a result of these requirements, we propose the Workflow Responsive resource Provisioning and Scheduling (WRPS) algorithm for scientific workflows in clouds. Our solution finds a balance between making dynamic decisions to respond to changes in the environment and planning ahead to produce better schedules. It aims to minimise the overall cost of the utilised infrastructure while meeting a user-defined deadline. It is capable of deciding what compute resources to use considering heterogeneous VM types, when is the best time to lease them and when they should be released to avoid incurring in extra costs. Our simulation results demonstrate it is scalable in terms of the number of tasks in the workflow, it is robust and responsive to the cloud performance variability and it is capable of generating better quality solutions than the state-of-the-art algorithms.

II. Related Work

There have been several works since the advent of cloud computing that aim to efficiently schedule scientific workflows. Many of them are dynamic and are capable of adapting to changes in the environment. An example is the Dynamic Provisioning Dynamic Scheduling (DPDS) algorithm [4] in which the number of VMs is adjusted depending on how well they are being used by the application. Zhou et al. [5] also propose a dynamic approach designed to capture the dynamic nature of cloud environments from the performance and pricing point of view. Poola et al. [6] designed a fault tolerant dynamic algorithm based on the workflow's partial critical paths. The Partitioned Balanced Time Scheduling algorithm [7] estimates the optimal number of resources needed per billing period so that a deadline is met and the cost is minimised. Other dynamic algorithms include those developed by Xu et al. [8], Huu and Montagnat [9], and Oliveira et al. [10]. The main disadvantage of these approaches is their task-level optimisation strategy, which is a trade-off for their adaptability to unexpected delays.

On the other side of the spectrum are static algorithms. An example is the Static Provisioning Static Scheduling (SPSS) [4] algorithm. Designed to schedule a group of interrelated workflows (i.e. ensembles), it creates a provisioning and scheduling plan before running any task. Another example is the IaaS Cloud Partial Critical Path (IC-PCP) algorithm [11]. It is based on the workflow's partial critical paths and tries to minimise the execution cost while meeting a deadline constraint. Other examples include RDPS [12], DVFS-MODPSO [13] and EIPR [14]. In general, these algorithms are very sensitive to execution delays and runtime estimation of tasks, which is a trade-off for their ability to perform workflow-level optimisations and compare various solutions before choosing the best-suited one.

Contrary to fully dynamic or static approaches, our work combines both in order to find a better compromise between adaptability and the benefits of global optimisation. SCS [15] is an example of an algorithm attempting to achieve this. It has a global optimisation heuristic that allows it to find the optimal mapping of task to VM type. This mapping is then used at runtime to scale the resource pool in or out and to schedule tasks as they become ready for execution. Our approach is different to SCS in that the static component does not analyse the entire workflow structure and instead optimises the schedule of a subset of the workflow tasks. Moreover, our static component generates an actual schedule for these tasks rather than just selecting a VM type.

III. Application and Resource Models

We consider workflows modelled as Directed Acyclic Graphs (DAGs); that is, graphs with directed edges and no cycles or conditional dependencies. Formally, a workflow W is composed of a set of tasks $T = \{t_1, t_2, \ldots, t_n\}$ and a set of edges E. An edge $e_{ij} = (t_i, t_j)$ exists if there is a data dependency between t_i and t_j, case in which t_i is said to be the parent of t_j and t_j the child of t_i. Based on this, a child task cannot run until all of its parent tasks have completed and its input data is available in the corresponding compute resource. Also, a workflow is associated with a deadline δ_W, defined as a time limit for its execution. Additionally, we assume that the size of a task S_t is measurable in Million of Instructions (MIs) and that, for every task, this information is provided as input to the scheduler.

A pay-as-you go model where VMs are leased on-demand and are charged per billing period is considered. Any partial utilisation results in the VM usage being rounded up to the nearest billing period. We model a VM type, VMT, in terms of its processing capacity PC_{VMT} and cost per billing period C_{VMT}. We define PC_{VMT} in terms of the number of instructions the CPU can process per second, Million of Instructions per Second (MIPS). It is assumed that for every VM type, its processing capacity in MIPS can be estimated based on the information offered by providers.

Workflows process data in the form of files. A common approach used to share these files among tasks is to use a peer-to-peer (P2P) model in which files are transferred directly from the VM running the parent task to the VM running the child task. Another technique is to use a global shared storage such as Amazon S3 as a file repository. In this case, tasks store their output in the global storage and retrieve their inputs from the same. We consider the latter model based on the advantages it offers. Firstly, the data is persisted and hence, can be used for recovery in case of failures. Secondly, it allows for asynchronous computation. In the P2P model, synchronous communication between

tasks means that VMs must be kept running until all of the child tasks have received the corresponding data. With a shared storage on the contrary, the VM running the parent task can be released as soon as the data is persisted in the storage system. This may not only increase the resource utilisation but also decrease the cost of VM leasing.

We assume data transfers in and out of the global storage system are free of charge, as is the case for products like Amazon S3, Google Cloud Storage and Rackspace Block Storage. As for the actual data storage, most cloud providers charge based on the amount of data being stored. We do not include this cost in the total cost calculation of neither our implementation nor the implementation of the algorithms used for comparison in the experiments. The reason for this is to be able to compare our approach with others designed to transfer files in a P2P fashion. Furthermore, regardless of the algorithm, the amount of stored data for a given workflow is most likely the same in every case or it is similar enough that it does not result in a difference in cost.

We acknowledge the existence of VM provisioning and deprovisioning delays and assume that the CPU performance of VMs is not stable [2]. Instead, it varies over time with its maximum achievable value being the CPU capacity advertised by the provider. In addition, we assume network congestion causes a variation in data transfer times [16]. The bandwidth assigned to a transfer depends on the current contention for the network link being used. In addition, we assume a global storage with an unlimited storage capacity. The rates at which it is capable of reading and writing data vary based on the number of processes currently writing or reading data from the system. Finally, the processing time of a task t on a VM of type VMT, PT_t^{VMT}, is defined as the sum of its execution time and the time it takes to read the input files from the storage and write the generated output files to it. Note that whenever a parent and a child task are running in the same VM, there is no need to read the child's input file from the storage.

IV. The WRPS Algorithm

This section describes the reasoning behind the WRPS heuristics as well as a detailed explanation of the algorithm.

A. Overview and Motivation

WRPS has dynamic and a static features. Its dynamicity lies in the fact that the scheduling decisions are made at runtime, every time tasks are released into an execution queue. This allows it to adapt to unexpected delays caused by poor estimates or by environmental changes such as performance variation, network congestion, and VM provisioning delays. The static component expands the ability of the algorithm from making decisions based on a single task (the next one in the queue) to making decisions based on a group of tasks. The purpose is to find a balance between the local knowledge of dynamic algorithms and the global knowledge of static ones. This is achieved by introducing the concept of *pipeline* and by statically scheduling all of the tasks in the execution queue at once. In this way, WRPS is able to make better optimisation decisions and find better quality schedules.

A *pipeline* is a common topological structure in scientific workflows and is simply a group of tasks with a one-to-one, sequential relationship between them. Formally, a pipeline P is defined as a set of tasks $T_p = \{t_1, t_2, \ldots, t_n\}$ where $n \geq 2$ and there is an edge $e_{i,i+1}$ between task t_i and task t_{i+1}. In other words t_1 is the parent of t_2, t_2 the parent of t_3, and so on. The first task in a pipeline may have more than one parent but it must only have one child task. All other tasks must have a single parent (the previous pipeline task) and one child (the next pipeline task). A pipeline is associated with a deadline δ_P which is equal to the deadline of the last task in the sequence. An example is shown in Figure 1a.

By identifying pipelines in a workflow, we can easily expand the view from a single task to a set of tasks that can be scheduled more efficiently as a group rather than on their own. To avoid communication and processing overheads as well as VM provisioning and deprovisioning delays, tasks in a pipeline are clustered together and are always assigned to run on the same VM. The reasons are twofold. Firstly, the tasks are sequential and are required to run one after the other. There is no benefit in terms of parallelisation on assigning them to different VMs. Secondly, the output file of a task becomes the input file of the next one, by running on the same VM, we avoid the cost and time of transferring these files in and out of the global storage.

The strategy used to schedule queued tasks is derived from the topological features of scientific workflows. Aside from pipelines, a workflow also has parallel structures composed of tasks with no dependencies between them. These tasks can run simultaneously and are generally found whenever data distribution or aggregation takes place. In data distribution [17] a tasks's output is distributed to multiple tasks for processing. In data aggregation [17] the output of multiple tasks is aggregated, or processed, by a single task. Figure 1a shows an example of each of these structures.

The parallel tasks in these structures can be either homogeneous (same type) or heterogeneous (different types). The case in which the tasks are homogenous is common in scientific workflows; examples of well-known applications with this characteristic are Epigenomics, SIPHT, LIGO, Montage, and CyberShake. Based on this, we devise a strategy to efficiently schedule homogeneous parallel tasks that are of the same size (MIs) and are at the same level in the DAG. When using a level-based deadline distribution heuristic, these tasks will also have the same deadline. As an example, consider the data aggregation case. All the parallel tasks have to finish running before the aggregation task can start, therefore they can be assigned the same deadline which would be equal to the time the aggregation task is due to start. Note that the case in which tasks are heterogeneous

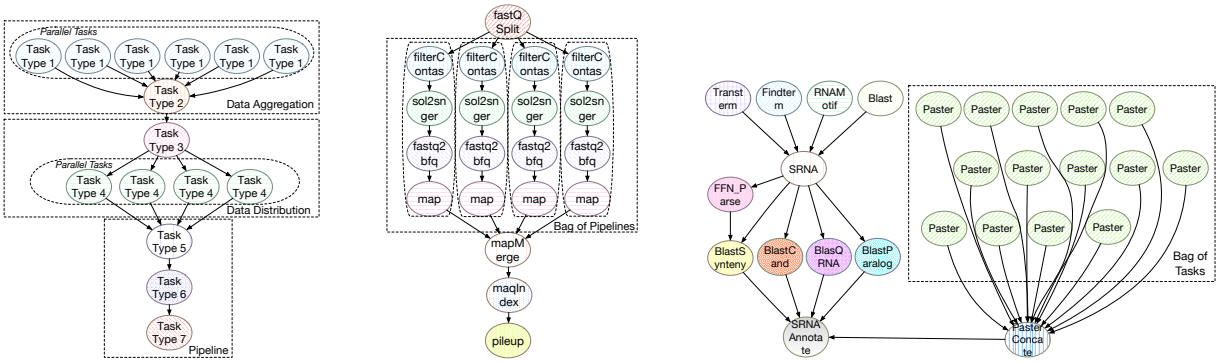


Figure 1. Scientific workflow examples. (a) Example of bag of tasks and three different topological structures found in workflows: data aggregation, data distribution and pipelines. (b) Epigenomics workfow. An example of a bag of pipelines is depicted in this figure. (c) SIPHT workflow.

and at different levels in the workflow is uncommon but yet possible. An example is the data distribution of the *SRNA* task in the SIPHT workflow, shown in Figure 1c. Also, there are other scenarios aside from distribution and aggregation where parallel tasks with the same properties can be found, however we focus on these as means for illustrating the motivation behind our scheduling policy.

The main static scheduling policy of WRPS consists then on grouping queued tasks of the same type and with the same deadline into bags. Two sample bags can be seen in Figure 1a, the first one is composed of all tasks of *Type 1* and the second one of all tasks of *Type 4*. Scheduling these bags of tasks is much simpler than scheduling a workflow. There are no dependencies, the tasks are homogenous, and have to finish at the same time. We model the problem of running these tasks before their deadline and with minimum cost as a variation of the unbounded knapsack problem and find an optimal solution using dynamic programming. The same concept is applied to pipelines, they are grouped into bags and scheduled in the same way as bags of tasks are. An example of a bag of pipelines is depicted in Figure 1b.

We have therefore designed an algorithm which is dynamic to a certain extent in order to adapt to unexpected delays product of the unpredictability of cloud environments but that also has a static component that enables it to generate better quality schedules and meet deadlines at lower costs. Moreover, it combines a heuristic-based approach with dynamic programming in order to be able to process large-scale workflows in an efficient and scalable manner. The details of WRPS are presented in Section IV-C.

B. The Unbounded Knapsack Problem

The unbounded knapsack problem (UKP) is an NP-hard combinatorial optimisation problem that derives from the problem of selecting the most valuable items to pack in a fixed-size knapsack. Given n items of different types, each item type $1 \leq i \leq n$ with a corresponding weight w_i and value v_i, the goal is to determine the number and type of items to pack so that the knapsack weight limit W is not exceeded and the total value of the items is maximised. Unlimited quantities of each item type are assumed.

Let $x_i \geq 0$ be the number of items of type i to be placed in the bag. Then UKP can be defined as

$$\text{maximise} \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \sum_{i=1}^{n} w_i x_i \leq W.$$

This problem can be solved optimally using dynamic programming by considering knapsacks of smaller capacities as subproblems and storing the best value for each capacity. Let $w_i > 0$, then a vector M can be defined where $m[w_i]$ is the maximum value that can be obtained with a weight less than or equal to w_i. In this way, $m[0] = 0$ and $m[w_i] = \max_{w_j \leq w_i}(v_j + m[w_i - w_j])$. The time complexity of this solution is $O(nW)$ as computing each $m[w_i]$ involves examining n items and there are W values of $m[w_i]$ to calculate. This running time is pseudo-polynomial as it grows exponentially with the length of W. Yet, there are several algorithms that can efficiently solve UKP. An example is the EDUK [18] algorithm which combines the concepts of dominance [19], periodicity [20], and monotonic recurrence [21]. Experiments performed by the authors demonstrate its scalability. For instance, for $W > 2 \times 10^8$, $n = 10^5$, and items with weights in the $[1, 10^5]$ range, the average running time was found to be 0.150 seconds.

C. Algorithm

WRPS first preprocesses the DAG by identifying the pipelines and by assigning a portion of the deadline δ_W to each task. To find the pipelines, tasks are first sorted in topological order, in this way we ensure data dependencies are preserved. Afterwards, pipelines are built based on the following logic. For each sorted task that has not been processed, the algorithm recursively tries to build a pipeline that starts with that task. The base cases of the recursion happen when the processed task has no children, when it has more than one child or, when it has a single child

with more than one parent task. The recursive stage occurs when the processed task has strictly one child which at the same time has strictly one parent (the processed task). In this case the task is added to the pipeline and the recursion continues with its child task. Once a pipeline was identified and the recursion finishes, the process is repeated for the next unprocessed sorted task, this continues until all the tasks have been processed. A more detailed explanation of the recursive part of the algorithm is depicted in Algorithm 1.

Algorithm 1 Find a pipeline recursively

```
1: procedure FINDPIPELINE(Task t, Pipeline p)
2:     if t.children.size > 1 OR t.children.size = 0 OR
3:     t.children[0].parents.size > 1 then
4:         if p.tasks.size > 0 then
5:             p.addTask(t)
6:         end if
7:         return
8:     end if
9:     p.addTask(t)
10:    findPipeline(t.children[0], p)
11: end procedure
```

For the deadline distribution, the algorithm first calculates the earliest finish time of all tasks defined as $eft_t = \max_{p \in t.parents}\{eft_p\} + PT_t^{VMT}$. The slowest VMT is used to calculate the task processing times. In this way, they can only improve if different VM types are used. However, if using the slowest VM type means not being able to meet the deadline, then the next fastest VM type is used to estimate runtimes and so on. Afterwards, the spare time, defined as the difference between the deadline and the earliest finish time of the workflow ($\delta_W - \max_{t \in W}\{eft_t\}$) is calculated and divided between the workflow levels based on the number of tasks they have. Finally, each task is assigned its deadline $\delta_t = \max_{p \in t.parents}\{\delta_p\} + PT_t^{VMT} + t.level.spare$.

Once a DAG is preprocessed the task scheduling begins. During the first iteration, all the entry tasks (those with no parent tasks) become ready for execution and are placed in a scheduling queue.Tthese tasks are scheduled and after they finish their execution, their child tasks are released onto the queue. This process is repeated until all of the workflow tasks have been successfully executed. To schedule the tasks in the queue, tasks are first grouped into bags of tasks and bags of pipelines. A bag of tasks *bot* is defined as a group of one or more tasks T_{bot} that can run in parallel. All of the tasks in a bag share the same deadline δ_{bot}, are of the same type τ_{bot}, and are not part of a pipeline. Formally, $bot = (T_{bot}, \delta_{bot}, \tau_{bot})$. The definition of bag of pipelines *bop* is similar but instead of a group of tasks, the bag contains one or more pipelines P_{bop} that are parallelisable. The tuple $bop = (P_{bop}, \delta_{bop})$, where δ_{bop} is the deadline the pipelines in the bag have in common, formally defines the concept.

To find the sets of bags of tasks $BoT = \{bot_1, \ldots, bot_n\}$ and bags of pipelines $BoP = \{bop_1, \ldots, bop_n\}$, each task in the queue is processed in the following way. If the task does not belong to a pipeline, then it is placed in the bot_i that contains tasks of the same type and with the same deadline. If no such bot_i exists, a new bag is created and the task assigned to it. If, on the other hand, the task belongs to a pipeline, the corresponding pipeline is placed in the bop_i which contains pipelines with the same deadline and types of tasks. If there is no bop_i with these characteristics, a new bag is created with its only element being the given pipeline.

Once the sets BoP and BoT are created, we proceed to schedule them. Both types of bags are scheduled using the same policy, with the only difference being that tasks in a pipeline must be treated as a unit. We explain the heuristic using BoT, note however that the same rules apply when scheduling BoP. To schedule BoT, we repeat the following process for each bag $bot_i \in BoT$ that has more than one task (bags with a single element are treated as a special case and scheduled accordingly). First, WRPS tries to reduce the size of the bag and reuse already leased VMs by assigning as many tasks as possible to idle VMs. The number of tasks mapped to a free VM is determined by the number of tasks that can finish before their deadline and before the next billing period of the VM. In this way, wastage of already paid CPU cycles is reduced without affecting the makespan of the workflow. After this, a resource provisioning plan is created for the remaining tasks in the bag.

To generate an efficient resource provisioning plan, WRPS must explore different solutions using different VM types and compare their associated costs. We achieve this and find the optimal combination of VMs that can finish the tasks in the bag in time with minimum cost by formulating the problem as a variation of UKP and solve it using dynamic programming. A knapsack item is defined by its type, weight, and value. For our problem, we define a scheduling knapsack item $SKI_j = (VMT_j, NT_j, C_j)$ where the item type corresponds to a VM type VMT_j, the weight is the maximum number of tasks NT_j that can run in a VM of type VMT_j before their deadline, and the value is the associated cost C_j of running NT_j tasks in a VM of type VMT_j. Additionally, we assume there is an unlimited quantity of VMs of each type that can be potentially leased and define the knapsack weight limit as the number of tasks in the bag, that is, $W = |T_{bot}|$. The goal is to find a set of items SKI so that their combined weight (the total number of tasks) is at least as large as the knapsack weight limit (the number of tasks in the bag) and whose combined value is minimum (the cost of running the tasks is minimum). Formally, the problem of scheduling a bag of tasks is expressed as

$$\text{minimise} \sum_{i=1}^{n} C_i \times x_i \quad \text{subject to} \sum_{i=1}^{n} NT_i \times x_i \geq |T_{bot}| \,.$$

An example on how the resource provisioning plan is generated for a bag to tasks is shown in Figure 2. Assume VM types, VMT_1 and VMT_2. The former can process 1 $instructions/sec$ at a cost of $\$1/minute$ and the latter 10

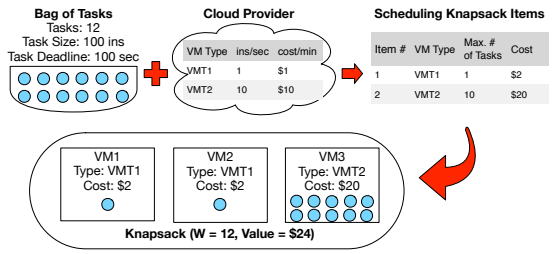


Figure 2. Example of a scheduling plan for a bag of tasks

$instructions/sec$ at a cost of $\$10/minute$. Consider a bag of 12 tasks, each task has a size of 100 instructions and a deadline of 100 seconds. The first knapsack item would be $SKI_1 = (VMT_1, 1, \$2)$. $NT_1 = 1$ as running a task in a VM that can process 1 $instructions/sec$ takes 100 seconds, with a deadline of 100 seconds, this means only one task can run on it before the deadline. $C_1 = \$2$ since the VM billing period is 60 seconds, this means that to use it for 100 seconds, two billing periods need to be paid for. Following the same reasoning, the second knapsack item would be $SKI_2 = (VMT_2, 10, \$20)$. The optimal combination of items to pack is then 2 items of type SKI_1 and 1 item of type SKI_2. This means leasing 2 VMs of type VMT_1 and running 1 task on each and leasing 1 VM of type VMT_2 and running 10 tasks on it, for a total cost of \$24.

After solving the UKP problem, a resource provisioning $RP_i^{bot} = (VMT_i, numVM_i, NT_i)$ is obtained for each VM type. It indicates the number of VMs of type VMT_i to use ($numVM_i$) and the maximum number of tasks to run on each VM (NT_i). In rare cases in which there are no VM types that can finish the tasks on time, a provisioning plan of the form $RP_{bot}^{fastest} = (VMT_{fastest}, W, 1)$ is created. This indicates that a VM of the fastest type must be leased for each task in the bag so that they can run in parallel and finish as early as possible. Then, for each RP_i^{bop} for which $numVM_i > 0$, WRPS tries to find a VM of type VMT_i which has already been leased and is free to use. If it exists, then NT_i tasks from the bag are scheduled on to it. In this way we reduce cost by using already paid for time slots and avoid long provisioning delays of newly leased VMs. If there is no free VM of the required type, a new one is provisioned and NT_i tasks are scheduled on to it.

We consider the case of a bags with a single task as a special one. Single tasks are scheduled on free VMs if they can finish the task on time and before their current billing period finishes. If there is no free VM that can achieve this, then a new VM of the type that is capable of finishing the task by its deadline at the cheapest cost is provisioned and the task scheduled to it. If no VM type can finish by the deadline, the fastest available VM type is used. The pseudo-code for the scheduling of BoT is shown in Algorithm 2.

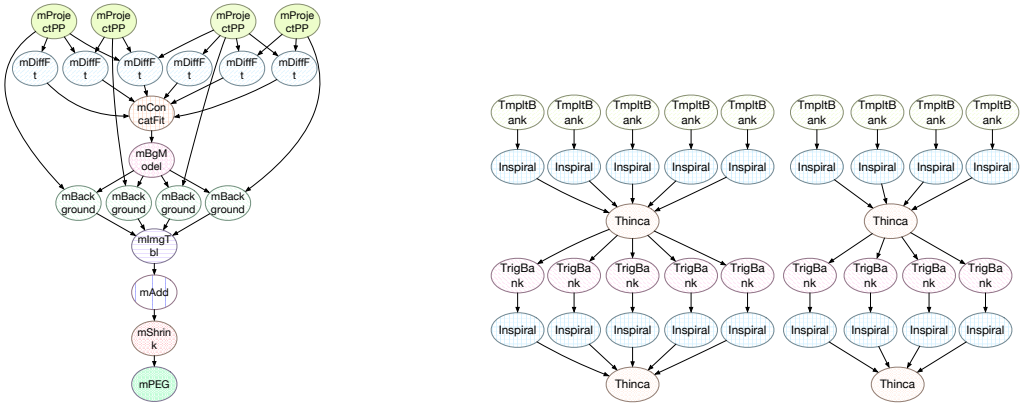
WRPS continuously adjusts the deadline distribution to reflect the actual finish time of tasks. If a task finishes earlier than expected, all of the remaining tasks will have more time

Algorithm 2 BoT scheduling

```
1: procedure ScheduleBoT(BoT)
2:     for all bot ∈ BoT do
3:         if bot.tasks.size > 1 then
4:             RP^bot = UKPBasedProvisioningPlan(bot)
5:             for all RP_i^bot = (VMT_i, numVM_i, NT_i) ∈ RP^bot do
6:                 for k = 0; k < numVM_i; k++ do
7:                     nTasks = min{NT_i, bot.size}
8:                     tasks = {t_1, ..., t_nTasks | t_i ∈ bot.tasks}
9:                     remove tasks from bot.tasks
10:                    vm = findFreeVM(VMT_i)
11:                    if vm == null then
12:                        vm = provisionNewVM(VMT_i)
13:                    end if
14:                    scheduleTasks(tasks, vm)
15:                end for
16:            end for
17:        else
18:            task = bot.taks[0]
19:            vm = findFreeVMForTask(task, task.deadline);
20:            if vm == null then
21:                vm = provisionNewVM(VMT_i)
22:            end if
23:            scheduleTask(task, vm)
24:        end if
25:    end for
26: end procedure
```

to run and cost can be potentially reduced. When a task finishes later than expected, the deadline of all remaining tasks is updated to avoid delays that could lead to the overall deadline being missed. WRPS also has a rescheduling mechanism that enables it to deal with unexpected delays encountered while running a bag of tasks (or pipelines). Since multiple tasks are statically assigned to a VM, a delay in the execution of one task will have an impact on the actual finishing time of the other ones. To mitigate this effect, when a task belonging to a bag finishes after its deadline on VM_i, then the tasks in the execution queue of VM_i are analysed in the following way. If all of the tasks remaining in the queue of VM_i can finish by their updated deadline then no action is taken. If VM_i cannot finish its queued tasks on time, then the tasks are released back into the scheduling queue so that they can be rescheduled based on their updated deadline.

As mentioned earlier, the set of bags of pipelines BoP is scheduled using the same strategy used for BoT. Just as tasks, pipelines have a deadline and a size (the aggregated size of all the pipeline tasks). Thus, we can apply the same scheduling heuristic and model the problem as a variation of UKP with a slight difference in the definition of a knapsack item. For pipelines, $SKI_j = (VMT_j, NP_j, C_j)$, where an item's weight NP_j, is equal to the maximum number of pipelines a VM type can finish before their deadline. The rescheduling strategy is also similar to that of tasks, except that whenever a task in a pipeline is delayed, the remaining pipeline tasks are left to finish in the VM while other pipelines are rescheduled based on their updated deadline. Once again, bags with a single pipeline are treated as a special case and are assigned to free VMs if they can finish



(a) (b)

Figure 3. (a) Montage workflow. (b) LIGO workflow.

Table I
VM TYPES BASED ON GOOGLE COMPUTE ENGINE OFFERINGS

Name	Memory	Google Compute Engine Units	Price per Minute
n1-standard-1	3.75GB	2.75	$0.00105
n1-standard-2	7.5GB	5.50	$0.0021
n1-standard-4	15GB	11	$0.0042
n1-standard-8	30GB	22	$0.0084

them by their deadline or to a newly leased VM of the type that can finish them by their deadline at minimum cost.

Finally, WRPS shuts down a VM if its use is approaching the next billing period and there are no tasks assigned to it. An estimate of the VM deprovisioning delay is used to ensure the VM shutdown request is sent early enough so that it stops being billed before the current billing period ends.

V. PERFORMANCE EVALUATION

WRPS was evaluated using four well-known workflows from various scientific areas: Montage from the astronomy field, Epigenomics and SIPHT from the bioinformatics domain, and LIGO from the astrophysics area. The Epigenomics and SIPHT structures are shown in Figure 1 while LIGO and Montage are depicted in Figure 3. Each of these workflows has different topological structures and different data and computational characteristics. Their description and characterisation is presented by Bharathi et al. [17].

Two algorithms were used to evaluate the quality of the schedules produced by WRPS. The first one is the Static Provisioning Static Scheduling (SPSS) [4] algorithm which assigns sub-deadlines to tasks and schedules them onto existing or newly leased VMs so that cost is minimised. It was designed to schedule a group of interrelated workflows but it can easily be adapted to schedule a single one. It was chosen as it is a static algorithm capable of generating high-quality solutions. Its drawback is its inability to meet deadlines when unexpected delays occur. However, we are still interested in comparing WRPS to SPSS when both are able to meet the deadline constraints. Also, by comparing them, we are able to validate our solution's adaptability and demonstrate how when other algorithms fail to recover from unexpected delays WRPS succeeds in doing so.

The second algorithm is SCS [15], a state-of-the-art dynamic algorithm that has an auto-scaling mechanism that allocates and deallocates VMs based on the current status of tasks. It determines the most cost-efficient VM type for each task and creates a *load vector*. This load vector is updated every scheduling cycle and indicates how many VMs of each type are needed in order for the tasks to finish by their deadline with minimum cost. The purpose is to demonstrate how the static component of WRPS allows it to produce better quality schedules than SCS.

An IaaS provider offering a single data centre and four types of VMs was modelled. The VM type configurations are based on the Google Compute Engine offerings and are shown in Table I. A billing period of 60 seconds was used, as offered by providers such as Google Compute Engine and Microsoft Azure. For all VM types, the provisioning delay was set to 30 seconds [22] and the deprovisioning delay to 3 seconds [3]. CPU performance variation was modelled after the findings by Schad et al. [2]. The performance of a VM is degraded by at most 24% based on a normal distribution with a 12% mean and a 10% standard deviation. A network link's total available bandwidth is shared between all the transfers using the link at a given point in time. This bandwidth allocation was done using the progressive filling algorithm [23] to model congestion and data transfer time degradation. A global shared storage with a maximum reading and writing speeds was also modelled. The reading speed achievable by a given transfer is determined by the number of processes currently reading from the storage, the same rule applies for the writing speed. In this way, congestion in the storage system is simulated.

Workflows with approximately 1000 tasks were used for the evaluation. We acknowledge that the estimation of task sizes might not be 100% accurate and hence, introduce in our simulation a variation of $\pm 10\%$ to the size of each task based on a uniform distribution. The experiments were conducted

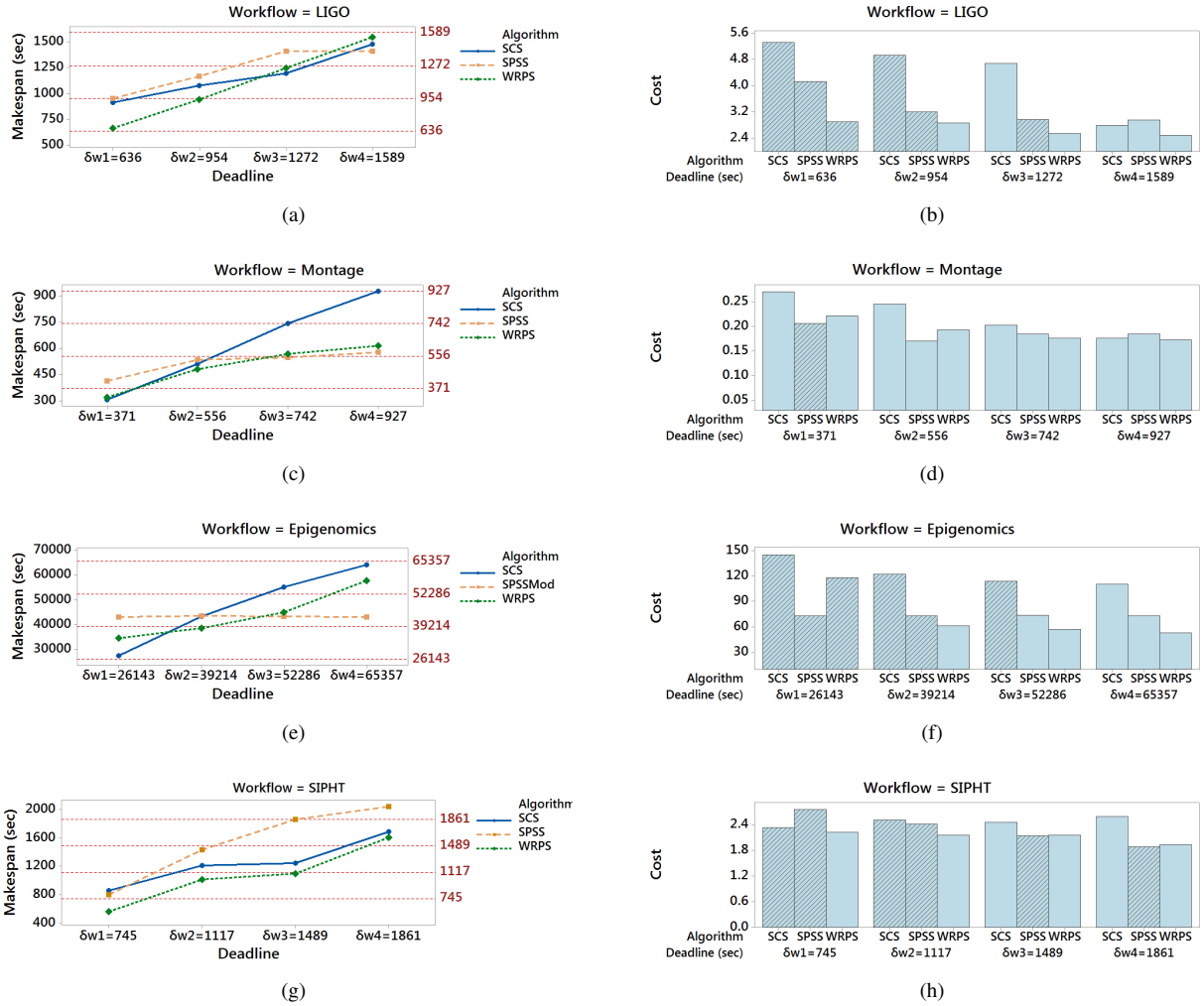


Figure 4. Makespan and cost experiment results. The reference lines in the makespan line plots indicate the four deadline values. The stripped bars in the cost bar charts indicate the deadline was not met for the corresponding deadline value. (a) LIGO makespan line plot. (b) LIGO cost bar chart. (c) Montage makespan line plot. (d) Montage cost bar chart. (e) Epigenomics makespan line plot. (f) Epigenomics cost bar chart. (g) SIPHT makespan line plot. (h) SIPHT cost bar chart.

using four different deadlines, δ_{W1} being the strictest one and δ_{W4} being the most relaxed one. For each workflow, δ_{W1} is equal to the time it takes to execute the tasks in the critical path plus the time it takes to transfer all the input files into the storage and the output files out of it. The remaining deadlines are based on δ_{W1} and an interval size $\delta_{int} = \delta_{W1}/2$: $\delta_{W2} = \delta_{W1} + \delta_{int}$, $\delta_{W3} = \delta_{W2} + \delta_{int}$, and $\delta_{W4} = \delta_{W3} + \delta_{int}$. The results displayed are the average obtained after running each experiment 20 times.

A. Results and Analysis

1) Makespan and Cost Evaluation: The average makespan and cost obtained for each of the workflows is depicted in Figure 4. The reference lines in the makespan line plots correspond to the four deadline values used for each workflow. Evaluating the makespan and cost with regards to this value is essential as the main objective of all the algorithms is to finish before the given deadline. The dashed bars in the cost bar charts indicate that the algorithm failed to meet the corresponding deadline.

For LIGO, δ_{W1} proves to be to tight for any of the algorithms to finish on time. However, the difference between the makespan obtained by WRPS and the deadline is marginal. Additionally, WRPS generates the cheapest schedule in this case. The second deadline is still not relaxed enough for SCS or SPSS to achieve their goal, however, WRPS demonstrates its adaptability and ability to generate cheap schedules by being the only one to finish its execution before the deadline and with the lowest cost. For the remaining deadlines, δ_{W3} and δ_{W4}, both SCS and WRPS are capable of meeting the constraint, in both cases SCS has a slightly lower makespan but WRPS has a lower cost. SPSS is only capable of meeting the most relaxed deadline, and in this case, WRPS outperforms it in terms of execution cost. Overall, WRPS

meets the most deadlines and in all of the cases achieves better quality schedules with the cheapest costs.

In the case of Montage, SCS and WRPS meet all of the deadlines with WRPS consistently generating cheaper schedules. SPSS fails to meet the tightest deadline but succeeds in meeting δ_{W2}, δ_{W3}, and δ_{W4}. Its success in meeting three out of four deadlines may be explained by the fact that most of the tasks in the Montage application are considered small and require low CPU utilisation, leading to a potentially low CPU performance variation impact on the static schedule. In the case of δ_{W2}, SPSS proves its ability to generate cheap schedules and performs better than its counterparts; although WRPS has a lower makespan in this case, its cost is slightly higher than that of SPSS. For δ_{W3} and δ_{W4} however, WRPS succeeds in generating cheaper solutions than its static counterpart.

The three algorithms fail to meet δ_{W1} of the Epigenomics workflow, the closest makespan to the deadline is achieved by SCS, followed by WRPS and finally SPSS. WRPS is the only algorithm capable of meeting δ_{W2} and still achieves the lowest cost. The third deadline constraint is met by SPSS and WRPS, with WRPS once again outperforming SPSS in terms of cost. Finally, as the deadlines become relaxed enough, the three algorithms succeed in meeting the deadline and WRPS does it with the lowest cost. The high deadline miss percentage of SCS and SPSS in this case is due to the high-CPU nature of the Epigenomics tasks, meaning that CPU performance degradation will have a significant impact on the execution time of tasks causing unexpected delays.

SIPHT is an interesting application to evaluate WRPS with due to its topological features. As mentioned earlier, it has data distribution and aggregation structures in which the parallel tasks differ on their type. The results demonstrate that even in cases like this, WRPS remains responsive and efficient. It succeeds in meeting all the deadlines with the lowest makespan and with the lowest cost amongst the algorithms that meet the constraint. The large number of files that need to be transferred when running this workflow lead to SPSS struggling to recover from the lower data transfer rates due to network congestion and hence failing to meet the four deadlines. Even SCS fails to adapt to these delays on time and fails to meet the three tightest deadlines.

Overall, WRPS is the most successful algorithm in meeting deadlines. On average, it succeeds in meeting the constraint in 87.5% of the cases while SCS succeeds in 56.25% and SPSS on 37.5%. These results are inline with what was expected of each algorithm. The static approach is not very efficient in meeting the deadlines whereas the dynamism in WRPS and SCS allows them to accomplish their goal more often. The experiments also demonstrate the efficiency of WRPS in terms of its ability to generate low cost solutions. It outperforms SCS an SPSS as in all of the scenarios except one (Montage workflow, δ_{W2}); WRPS achieves the lowest cost when compared to those algorithms that met

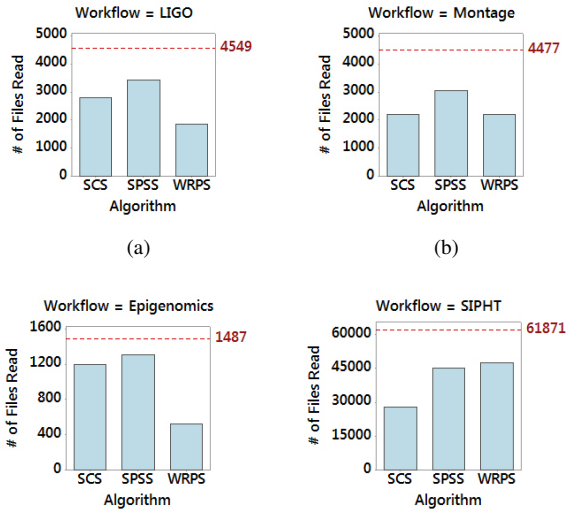


Figure 5. Average number of files read from the storage by each algorithm. The reference lines indicate the total number of files required as input by the given workflow. (a) LIGO. (b) Montage. (c) Epigenomics. (d) SIPHT.

the deadline. Another desirable characteristic of WRPS that can be observed from the results is its ability to consistently increase the time it takes to run the workflow and reduce the cost as the deadline becomes more relaxed. The importance of this relies in the fact that many users are willing to trade-off execution time for lower costs while others are willing to pay higher costs for faster executions. The algorithm needs to behave within this logic in order for the deadline value given by users to be meaningful.

2) Network Usage Evaluation: Network links are well-known bottlenecks in Cloud environments. For instance, Jackson et al. [16] report a data transfer time variation of up to 65% in Amazon EC2. Hence, as means of reducing the sources of unpredictability and improving the performance of workflow applications, it is important for scheduling algorithms to try to reduce the amount of data transferred through the cloud network infrastructure. In this section, we evaluate the number of files read from the global storage by each of the algorithms. Recall that a task does not need to read from the storage system whenever the input files it requires are already available in the VM where it is running.

The bar charts in Figure 5 show the average number of files read across the four deadlines for each workflow and algorithm. The reference line indicates the total number of input files that are required by the workflow tasks. By scheduling pipelines in a single VM and by running as many tasks or pipelines from the same bag in a single VM, WRPS is successful in reducing by 50% or more the number of files read from the storage. In fact, WRPS reads the least amount of files when compared to SCS and SPSS in the cases of the LIGO and Epigenomics workflow. The files read from the storage are reduced by 58% in the LIGO case and by 75% in

the Epigenomics case. For the Montage workflow, SCS and WRPS achieve a similar performance and reduce the number of files by approximately 50%. Finally, even though reduced by 23%, WRPS is not as successful in reducing the number of files as SCS and SPSS for the SIPHT workflow. The lack of pipelines and bags of tasks in the SIPHT application are the main cause for this and as a future work we would like to explore and develop new heuristics so that WRPS is capable of scheduling these type of workflows more efficiently.

VI. Conclusions

WRPS, a responsive resource provisioning and scheduling algorithm for scientific workflows in clouds capable of generating high quality schedules was presented. It has as objectives minimising the overall cost of using the cloud infrastructure while meeting a user-defined deadline. The algorithm is dynamic to a certain extent to respond to unexpected delays and environmental dynamics common in cloud computing. It also has a static component that allows it to find the optimal schedule for a group of workflow tasks, consequently improving the quality of the schedules it generates. By reducing the workflow into bags of homogeneous tasks and pipelines that share a deadline, we are able to model their scheduling as a variation UKP and solve it in pseudo-polynomial time using dynamic programming.

The simulation experiments show that our solution has an overall better performance than state-of-the-art algorithms. It is successful in meeting deadlines under unpredictable situations involving performance variation, network congestion and inaccurate task size estimations. It achieves this at low costs, even lower than fully static approaches which have the ability of using the entire workflow structure and comparing various solutions before the workflow execution.

References

[1] J. D. Ullman, "Np-complete scheduling problems," *J. Comput. System Sci.*, vol. 10, no. 3, pp. 384–393, 1975.

[2] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[3] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. Int. Conf. Cloud Comput. (CLOUD)*, 2012.

[4] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2012.

[5] A. C. Zhou, B. He, and C. Liu, "Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds," *IEEE Trans. Cloud Comput.*, vol. PP, no. 99, pp. 1–1, 2015.

[6] D. Poola, K. Ramamohanarao, and R. Buyya, "Fault-tolerant workflow scheduling using spot instances on clouds," *Procedia Comput. Sci.*, vol. 29, pp. 523–533, 2014.

[7] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng, "Cost optimized provisioning of elastic resources for application workflows," *Future Generation Comput. Syst.*, vol. 27, no. 8, pp. 1011–1026, 2011.

[8] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing," in *Proc. Int. Symp. Parallel Distrib. Processing Applicat. (ISPA)*, 2009.

[9] T. T. Huu and J. Montagnat, "Virtual resources allocation for workflow-based applications distribution on a cloud infrastructure," in *Proc. Int. Conf. Cluster, Cloud Grid Comput. (CCGrid)*, 2010.

[10] D. de Oliveira, K. A. Ocaña, F. Baião, and M. Mattoso, "A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds," *J. Grid Comput.*, vol. 10, no. 3, pp. 521–552, 2012.

[11] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 158–169, 2013.

[12] Z. Wu, Z. Ni, L. Gu, and X. Liu, "A revised discrete particle swarm optimization for cloud workflow scheduling," in *Proc. Int. Conf. Computational Intell. Security (CIS)*, 2010.

[13] S. Yassa, R. Chelouah, H. Kadima, and B. Granado, "Multi-objective approach for energy-aware workflow scheduling in cloud computing environments," *Sci. World J.*, 2013.

[14] R. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1787–1796, 2014.

[15] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Performance Comput., Newt., Storage Anal. (SC)*, 2011.

[16] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Proc. Int. Conf. Cloud Comput. Technol. and Sci. (CloudCom)*, 2010.

[17] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proc. Workshop Workflows Support Large-Scale Sci. (WORKS)*, 2008.

[18] R. Andonov, V. Poirriez, and S. Rajopadhye", "Unbounded knapsack problem: Dynamic programming revisited," *European J. Oper. Research*, vol. 123, no. 2, pp. 394 – 407, 2000.

[19] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting stock problem-part ii," *Oper. Research*, vol. 11, no. 6, pp. 863–888, 1963.

[20] P. Gilmore and R. Gomory, "The theory and computation of knapsack functions," *Oper. Research*, vol. 14, no. 6, pp. 1045–1074, 1966.

[21] R. Andonov and S. Rajopadhye, "A sparse knapsack algo-tech-cuit and its synthesis," in *Proc. Int. Conf. Appl. Specific Array Processors*, 1994.

[22] S. Stadill, "By the numbers: How google compute engine stacks up to amazon ec2," https://gigaom.com/2013/03/15/by-the-numbers-how-google-compute-engine-stacks-up-to-amazon-ec2/.

[23] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-Hall International New Jersey, 1992.