The Journal of Systems and Software 118 (2016) 101–114

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Dynamic auto-scaling and scheduling of deadline constrained service workloads on IaaS clouds

Elias De Coninck*, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Pieter Simoens, Bart Dhoedt

Department of Information Technology, Ghent University - iMinds, Technologiepark-Zwijnaarde 15, Ghent B-9052, Belgium

ARTICLE INFO

Article history:
Received 22 October 2015
Revised 4 May 2016
Accepted 5 May 2016
Available online 6 May 2016

Keywords:
Cloud computing
Deadline constrained workflow scheduling
Dynamic resource allocation

ABSTRACT

Cloud systems are becoming attractive for many companies. Rather than over-provisioning the privately owned infrastructure for peak demands, some of the work can be overspilled to external infrastructure to meet deadlines. In this paper, we investigate how to dynamically and automatically provision resources on the private and external clouds such that the number of workloads meeting their deadline is maximized. We specifically focus on jobs consisting of multiple interdependent tasks with a priori an unknown structure and even adaptable at runtime. The proposed approach is model-driven: knowledge on the job structure on the one hand; and resource needs and scaling behavior on the other hand. Information is built up based on monitoring information and simulated 'what-if'-scenarios. Using this dynamically constructed job resource model, the resources needed by each job in order to meet its deadline is derived. Different algorithms are evaluated on how the required resources and jobs are scheduled over time on the available infrastructure. The evaluation is carried out using synthetic workloads.

1. Introduction

The availability of reliable public cloud infrastructures has shifted large amounts of computational work from privately owned clusters to these public infrastructures. However, for security concerns and cost reasons, it is often beneficial to consider a hybrid cloud solution, where a privately owned system seamlessly interoperates with one or more external infrastructure services Sotomayor et al. (2009). The normal workload and/or security sensitive jobs are executed on the private infrastructure, while peak loads are offloaded to the public part. The private cloud investment cost is thus limited to the capacity needed to handle average workloads, without having to compromise on flexibility to respond to peak demands.

In this paper we propose the design of a system supporting the dynamic resource allocation in multiple clouds for jobs, with an uncertain adaptable task structure and an unknown execution time, needing completion before a given deadline. We consider workflow-oriented jobs, consisting of multiple tasks ordered in a directed acyclic graph (DAG) modelling the input/output dependency. Our system will schedule jobs, dynamically adapting the number of allocated resources in order to meet the deadlines of all jobs without knowing the DAG structure itself and without any information of the execution time.

The makespan of a single job depends on the number of resources allocated for this job and the number of components running in parallel, which is decided by the framework based on the given deadline. All jobs are scheduled onto a deadline ordered job queue and the job scheduler is responsible for selecting the correct job(s) to execute. The main contribution of this paper is to provide a workflow management system that supports existing scheduling algorithms with learned knowledge on job type, job structure, job makespan and available resources. Simulating jobs for a varying number of resources to accelerate the acquisition of more knowledge, allows our system's schedulers to cope with jobs having a priori an unknown execution time and with changes of the job's internal structure over time.

Our proposed system has been developed on top of OpenStack and AIOLOS Bohez et al. (2014) which allows on-demand provisioning of VMs on multiple cloud providers and distributing OSGi components. For each submitted job, our system follows six steps:

1. Queue the newly submitted job onto a deadline ordered job queue.
2. Look-up the knowledge in the model for the job type, allowing to derive the resource set required to meet the job's deadline.

* Corresponding author.
E-mail addresses: elias.deconinck@ugent.be, elias.deconinck@intec.ugent.be (E. De Coninck), tim.verbelen@ugent.be (T. Verbelen), bert.vankeirsbilck@ugent.be (B. Vankeirsbilck), steven.bohez@ugent.be (S. Bohez), pieter.simoens@ugent.be (P. Simoens), bart.dhoedt@ugent.be (B. Dhoedt).

http://dx.doi.org/10.1016/j.jss.2016.05.011
0164-1212/

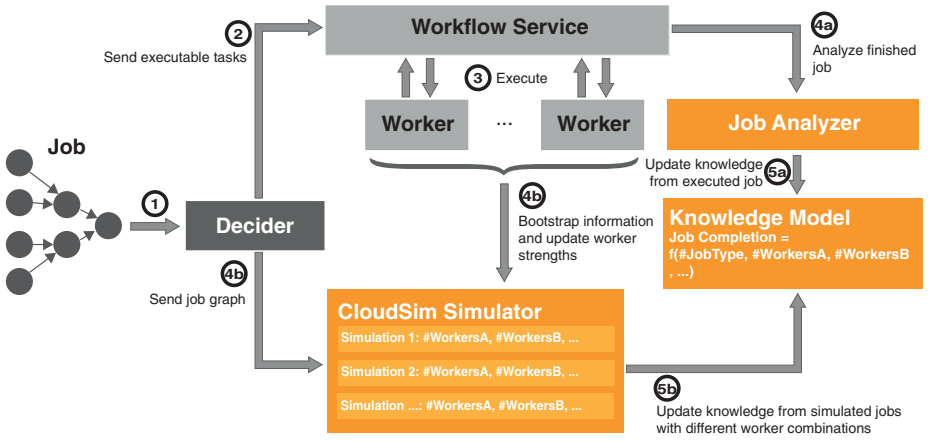


Fig. 1. Functional flow to increase knowledge from monitoring information and simulations. To measure infrastructure strength worker performance is monitored and bootstrapped into the simulations.

3. Allocate the required resources from an available private or public IaaS provider with spare capacity.
4. Schedule the tasks of each job on these allocated resources, such that the deadline of the job is met.
5. Simulation of the job execution for a number of varying resources, allowing to increase the knowledge model to improve future job planning.
6. Update knowledge model with monitoring information from the executed and simulated jobs.

After completion of each job, performance metrics are collected on the job completion time and the actual resource strength and usage. These metrics are used to update the knowledge model that the system maintains and will be used to more accurately plan and schedule future jobs of the same job type. Infrastructure performance metrics measure the capacity of VMs, which are used during job simulations to accurately update the knowledge model and to adapt for interference between VMs. Fig. 1 shows the basic flow of the system to build up knowledge.

This paper is structured as follows. Practical use case examples are provided in Section 2 to show the need for this framework. In Section 3, we describe the architecture of our framework showing how the major components of the system interact. Section 4 describes the scheduling algorithms investigated in combination with the resource allocation strategy. Section 5 explains the simulation component in detail, focussing on the additions made to an existing cloud simulation framework, needed to handle the scenarios relevant for this work. In Section 6, we highlight a number of design considerations of the framework and evaluation results are presented in Section 7. Section 8 discusses related work on scheduling workloads and criteria constrained workloads and finally Section 9 concludes this paper.

2. Application case studies

2.1. Manage, fuse and serve geospatial data

The first use case application is designed to manage, fuse and serve geospatial data. The goal is to efficiently store and retrieve geospatial data of a certain area to/from any database, service or file. The geographic information systems application (GIS) consists of three main components:

Server: the data server serves the geospatial data and relevant meta-data from a tile repository. The tile structure and depth are a priori unknown because areas with more detail are represented by more tiles. The data can be accessed by any interested component.

Client: the client accesses the data server to visualize a part of the map.

Engine: the engine is the computational worker to efficiently calculate and store the input data into the tile repository. A tile is the unit of geospatial data for storage and retrieval. It represents a patch of data corresponding to a location on the planet. Tiles exists at different levels of detail so visualization can be optimized by the requested detail level. In Fig. 2 the pyramid representation of detail levels is shown.

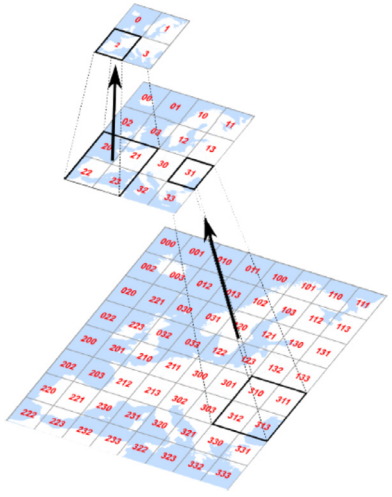


Fig. 2. A tile pyramid of Europe. A tile at a certain detail level spans exactly four tiles of a more detailed level.

To implement this application on top of the framework we need to represent the tile pyramid as a directed graph of tasks (see Fig. 3) and group them into a single job type. To enable the framework's functionality a Decider and one or more Workers are required. The Worker components can be automatically scaled by our framework.

GIS decider: the decider keeps track of the jobs internal structure. Tasks are submitted to the framework in order to fork, render or merge tiles. It decides if for a certain detail level four new tasks have to be spawned or if the tile has to be calculated.

Store: the store component stores tiles on the file system.

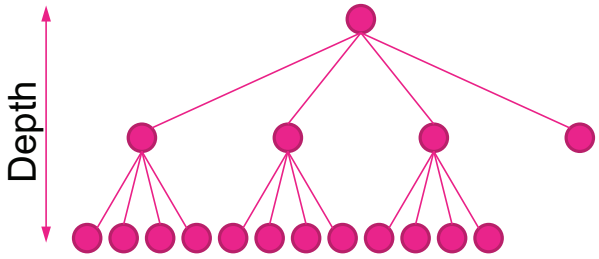


Fig. 3. Directed graph representation of a tile pyramid. The depth depends on the available details in a certain tile. Each tile can be split up in to four more detailed tiles on request.

Fusion task worker: is the worker which can execute fusion tasks. What it calculates is determined by the properties of the tasks. If the task represents the highest requested detail level it is classified as a leaf and the tile needs to be rendered and stored in the store component. Rendering this leaf tile from collected real-time data is the most compute-intensive task during job execution and is highly scalable. When the task needs more details it stops calculation and returns the finished task to the decider which in turn knows it has to create new tasks. And last if the task has descendants, which are already rendered, it merges the four descendants into one tile and stores the tile. As listed a fusion task has three possibilities which greatly affect the execution time of individual tasks. The proposed framework tries to generalize the entire job instead of the individual tasks.

The DAG structure for this job represented in Fig. 3 shows that the structure can change over time. When a job visualizes an other area the number of descendant tiles can be different and this changes the makespan of this job. The framework should be able to adapt to new structures and predict the makespan of these jobs.

2.2. Document processing use case

A well known application domain where scaling is necessary is the document processing use case. Many corporations focus on delivering a simple workflow which creates and archives invoices or other documents and sends them to clients if required. Many of these jobs are repeated frequently and possibly periodically. As an example we take a corporation that needs to send monthly invoices to their clients. At the end of each month such jobs need to be repeated and completed before a certain deadline. A single job consists of the following tasks:

- Fetch all clients we need to send invoices to this month.
- Load a personalized invoice template.
- Render personalized invoices for each client.
- Optional: send the invoice to the client (mail, e-mail, internet banking service, etc.).
- Zip and archive all invoices.

Each of these tasks depends on the previous task and can be executed in parallel for each client except for the last task, which depends on all tasks of all clients. In this use case the structure of the job is known before hand but the number of invoices is variable. Our framework needs to learn the execution time of these jobs and predict the required resources to complete these jobs before their deadline.

Document processing jobs tend to be less compute-intensive but have a known structure and is for many corporations more or less the same. We focus on a service provider who delivers a document processing service to multiple tenants all over the world. Every end of the month a peak load is expected and scaling up the number of workers will make sure the deadlines are still reached. In this case we target task distribution of many jobs with a shared structure but with different job settings. Each tenant has its own templates, business size and batch size setting.

3. System architecture

3.1. Terminology

We elucidate the terminology used in the architectural description below by the document processing use case described in Section 2.2. A corporation wants to compose client invoices and archive these before the end of each month. As described above this workflow consists of multiple **Task Types**. For clarity of the architectural terminology, we will discern two types only: 'invoice composition' and 'archival of composed files'. The composition of each individual invoice is a separate **Task** of the 'invoice composition' **Task Type**. The tasks are distinguished from each other by their type and their parameters (client *X* and creation date). When all invoices are composed, a single archival task archives all invoices and saves it to disk. The 'archival' task type requires the result of all 'invoice composition' tasks.

We distinguish workloads by the dependencies between task types. The corporation's workload has a single **Job Type**, 'archive client invoices', which is repeated as a **Job** every month. **Jobs** differ in the created client invoices, amount of invoices and the deadline date at the end of each month, but they will always create and store a single archive in the end. For each **Job Type** an application developer defines a corresponding **Decider**, which enforces the dependencies between tasks and gives the developer complete freedom on the workload structure.

Task Workers are components executing a particular task type. Multiple **Task Workers** of the same **Worker Type**, for example 'invoice composition', can be spawned simultaneously to allow parallel execution of these tasks. The duration of the job depends on the amount of workers allocated for each task type. To decrease the job makespan, our framework can instantiate more workers for highly parallel task types like 'invoice composition'. Workers are typically provided by the application developer, although we could envision an ecosystem of workers developed independently and being reused in multiple environments. Example workers are PDF generations, zip/unzip operations, image rendering, etc. Note that this terminology matches those of the Amazon Simple Workflow service Amazon simple workflow service (swf) (SWF), which however does not support multiple clouds and does not support simulations to speed up learning.

3.2. High level architecture

Fig. 4 illustrates the different components of the proposed deadline constrained workflow system. The **Workflow Service** acts as the main portal through which new jobs are submitted to the system. Based on the job description provided, the framework will instantiate workers of the appropriate type across multiple clouds. The number of workers per job is planned based on the deadline and a knowledge model, which is built through simulations and historical monitoring information by the **Capacity Planner**. Workers ask the workflow service for the next task to execute, and the order used to hand tasks to each worker is determined by the scheduler.

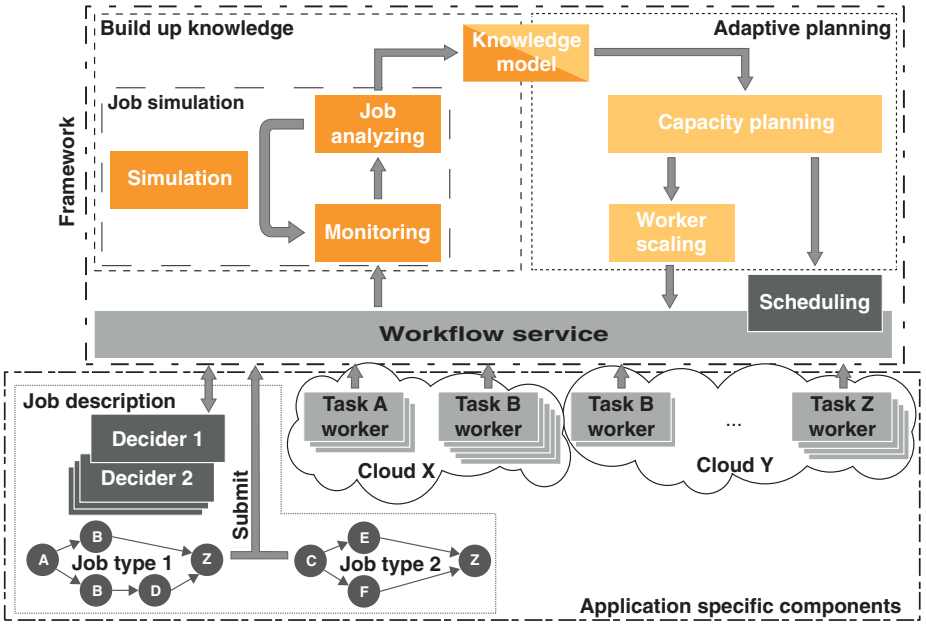


Fig. 4. Overview of the framework with the optional blocks: 'build up knowledge', 'adaptive planning' and 'job description'. The knowledge model is a shared component between two blocks.

3.3. Main framework components

3.3.1. Workflow service

The **Workflow Service** is the backbone of the framework. It handles the submission of new jobs and informs framework components about various events:

- Job added, starting, started, ended.
- Job queue changed.
- Task added, started, ended.
- Task execution simulation started, ended, failed.

The workflow service keeps track of all active tasks and jobs. Tasks can time out, in which case they will be resubmitted by the workflow service so the job can still complete. Resubmitting timed out tasks will increase the duration of a job and will be taken into account by the knowledge model. Multiple successive timed out tasks will eventually cause the current job execution to fail.

3.3.2. Knowledge model

The **Knowledge Model** is a shared component between the job analyser and the capacity planner. The goal is to accumulate information about submitted jobs, in order to accurately predict the behavior of future jobs of the same type. We focus on the execution time of real or simulated jobs instead of the individual tasks to minimize the overhead and to generalize on a higher level. An individual task can accomplish multiple tracks based on its parameters or result of previous tasks and can therefore greatly impact the execution time of a task.

The model stores the information for each specific job type separately. More specifically, the model keeps track of the execution time as a function of the number of workers of each worker type. More formally:

$$f(\#Type\ A, \#Type\ B, \ldots, \#Type\ N) = job\ type\ execution\ time \quad (1)$$

Planning resources for a deadline constrained job comes then down to identifying the number of workers for each worker type, resulting in a completion time before the deadline.

The knowledge is obtained by monitoring information of actually completed jobs. To bootstrap the model, or to enrich it with more resource configurations, an analyzer component may decide to simulate a job with varying number of worker instances. In the beginning it will start many simulations based on one real job to make an estimation of this workload type. Later only missing entries or uncertainties are simulated. The **Monitoring** module examines job execution progress and sends the information to the analyzer who updates the model and may decide on additional simulations. The simulator is a component which can execute jobs on a simulated cloud environment. When knowledge is sparse the analyzer decides to run additional simulations to improve the data model.

3.3.3. Adaptive planning

The **Capacity Planner** (CP) determines the number of workers for each worker type, based on the information on previously executed or simulated workloads available in the **Knowledge Model** component. The CP informs the **Worker Scaler** on the required workers, which in turn will provision the resources to accommodate for the planned jobs.

The **Knowledge Model** is contacted by the CP component each time a new job is submitted. All data points with a lower quantile function value than the current deadline are evaluated based on the required resources. We further discuss this approach in Section 4.1.

3.3.4. Scheduling

This component is responsible for scheduling jobs and is contacted by the **Workflow Service** each time a worker requests additional work. This is done by selecting a task for the requesting worker type from the job which has the highest priority in the job queue. The ranking of jobs is determined by an algorithm which we will discuss in Section 4.2.

3.4. Life cycle

The framework's life cycle follows the steps listed below. From the moment a job is submitted until the job and all simulations are finished.

1. Submit new job with a deadline to the workflow service.

2. Workflow Service informs all listening components and starts the job.
3. Capacity is planned based on the job type and current knowledge in the model.
4. Worker scaler scales number of workers based on the planned capacity.
5. This job type's decider queues all tasks which can start based on the job structure to the workflow service.
6. Repeat until job is completed:
   - Idle Workers poll the workflow service for an available task.
   - Deciders listen to completed tasks to queue child-tasks to the workflow service.
7. Completed jobs are monitored, analyzed and simulated.
8. Knowledge model is updated with new simulations based on the previously executed job.

4. Algorithms

4.1. Planning resources

Resources are planned by consulting the model and estimating the number of workers for each type, satisfying the job's deadline. If multiple configurations satisfy this requirement, the solution resulting in the lowest cost (in terms of VMs needed) is selected. The knowledge model is structured as a key value map where the keys are job types and the values represent ordered data point lists.

A data point is a combination of the estimated execution time and a standard deviation value for a unique combination of workers. All data points are part of the $\mathbb{N}^n$ domain with n the number different worker types. The estimated execution time is an exponential moving average and the error is a exponential moving standard deviation calculated with equations in (2). Each data point represents a normal distribution for the execution time. For each job type a list of unique data points is ordered based on the data point's estimated execution time value (μ). The weight value w is a parameter that can be used by the developer to control the adaptation rate. Higher w values make the data point move quicker to new values.

$$\begin{aligned} diff &= x - \mu \\ incr &= w * diff \\ \mu &= \mu + incr \\ \sigma &= \sqrt{(1-w) * (\sigma^2 + diff * incr)} \end{aligned} \qquad \text{with } x = \text{new execution time and } w \in \mathbb{R}[0,1] \tag{2}$$

To satisfy a newly submitted job's deadline with a 90% confidence level we select all data points with a quantile function value $F^{-1}(p)$, with $p = 0.9$, (3) which is lower than the job's deadline. Afterwards the data point with the lowest cost is selected. The cost function (4) can be further extended to allow for different VM types but our experiments are limited to a single VM type.

$$\begin{aligned} \Phi^{-1} &= \sqrt{2}\ erf^{-1}(2p-1) \\ F^{-1}(p) &= \mu + \sigma\,\Phi^{-1}(p) \end{aligned} \quad \text{with } p \in \mathbb{R}[0,1] \tag{3}$$

$$\text{data point cost} = \sum_{x=A}^{N} (\#Type\,x * VM\ type\ cost) \tag{4}$$

Tables 1 and 2 show an example of such a key value map with their corresponding execution time. The job type structure is not mentioned because the model learns this structure by simulating all possible worker type combinations.

4.2. Deadline constrained scheduling

The goal of this algorithm is to determine the ordering of the jobs as they will be started by the framework.

Table 1
Example job types.

Job type ID	Description	#Tasks
1	Archive client invoices	101
2	Optimize hardware acoustics	760

Table 2
Partial knowledge model key value map from the example job types in Table 1. Highlighted rows show the best possible combination with the least workers for each job type.

Job Type ID	#Workers Type				Execution	Error
	compose invoice	optimize	archive	...	time μ [s]	σ [s]
1	1		1		1 015	100
1	5		1		215	12
1	5		5		215	12
1	100		1		25	10
1	200		1		25	10
2		1	1		15 900	2560
2		250	1		213	110
2		500	10		57	54
2		600	15		57	54

FIFO: selects each job in the submitted order. Using this approach ignores the deadline when selecting a job for execution. Hence, this approach gives no guarantees whatsoever about meeting deadlines, but is included as a worst case scenario for benchmarking purposes.

Earliest Deadline First: orders the jobs in ascending order of the deadline (first deadline comes first), and optimisations such as queue backfilling are not used. This algorithm can produce good results because the more jobs are submitted the more resources will be allocated. But when one of the jobs exceeds its deadline all subsequent jobs might exceed their deadline.

Rate Limited Scheduler: uses information of the capacity planner to know how many workers of each worker type each job needs. The jobs are ordered with EDF but only workers allocated for this job are used. This means that jobs further in the queue can start before the first job has completed. If the capacity is realistically planned jobs should complete before their deadline.

5. Job simulation with CloudSim

CloudSim (Calheiros et al., 2011) is a framework for modelling and simulating cloud computing infrastructures and services. This simulator is used to enhance the knowledge model for a particular job type, without incurring the cost of actually processing some of these workloads. The simulation workload is generated based on the job's DAG structure and uses the actual task execution times to initialize the CloudSim workload. This ensures us that the task lengths are distributed like it would be on the physical infrastructure. This also mitigates the requirement to take into account the file read/writes and task parameters. To relate the results on job completion time obtained through simulations to wall clock time, a scaling factor is estimated:

$$job\ wall-clock\ time = x * job\ simulated\ execution\ time$$

The scaling factor 'x' depends on the actual VM resources at hand, and the value is calculated and updated as an exponentially weighted moving average based on comparing simulated time and actual time for the same job (with the same number of workers of each type). The weight of the moving average is a fixed value between zero and one or a value automatically decreased from one to

a minimum value larger than zero. This weight value can be configured by the application developer. Our framework is currently designed to work with a single resource type, but could be further extended to multiple types by adding a scaling factor for each VM type.

5.1. Simulation based knowledge build up

The **Knowledge Model** assumes that for each worker type there is at least one worker and the maximum number of workers for each worker type is limited by the job structure. Therefore, the simulator tries to find the maximum number of workers for each type for which further increasing the number of workers does not result in a decrease in job completion time. In the case of bag-of-tasks jobs, this maximum equals the number of tasks in the job. In more complex job structures, this limit is determined by the amount of parallelism in each worker type. Gradually, the **Knowledge Model** replaces simulation results by information gathered through actually execution jobs for various numbers of workers. The following algorithms are considered to steer the job simulator, and the list of data points to be simulated.

Basic: In this heuristic, three random scenarios are simulated for each executed job. Each scenario has a single random worker type set to a new random value between one and twice the current number of workers.

Exhaustive with stop criterion: makes a complete model but adapts slowly to resource intensive jobs. The current knowledge is used to find and update missing data points. This will increase the knowledge each step and doesn't calculate redundant information. A certain path that does not improve the results of its predecessor by a certain factor triggers the algorithm to ignore further investigation of the branch. Algorithm 1 shows the pseudocode for this approach.

```
Data: job_type
Result: Collection of unique data points
improve_factor ⟵ 0.10;
all_data_points ⟵ data points for job_type;
new_data_points ⟵ ∅;
foreach data point dp in all_data_points do
    if dp has parent
    and getExecutionTime(dp) ∗ (1 + improve_factor) >=
    getExecutionTime(parent) then
        next dp;
    end
    foreach (worker_type , count ) pair in dp do
        new_dp ⟵ dp;
        new_dp.put(worker_type) = count + 1;
        if new_dp not in all_data_points
        and new_dp not in new_data_points then
            add new_dp as child of dp;
            append new_dp to new_data_points;
        end
    end
end
return new_data_points;
```

Algorithm 1: Determine the data points where the knowledge model can improve. Next simulations use these data points to further built up knowledge.

5.2. Supporting complex jobs in CloudSim

The most important adaptation made to CloudSim in order to support the type of jobs relevant for the framework described above, is the addition of jobs consisting of interdependent tasks. This addition has resulted in an extension to CloudSim (rather than changing core CloudSim classes). Main extensions include:

- Extension of the cloudlet-concept (being the CloudSim entity representing a task in the terminology of the proposed framework). By inheritance support for graph-like workload structures is added. Each cloudlet (task) has references to parent and child cloudlets.
- Extension of the datacenter-entity:
  - Keeping track of parent-child relationships between task.s
  - Extending the broker, taking into account these dependencies, only spawning tasks when these can be executed (i.e. all dependent tasks have been successfully completed).
  - Ensuring that tasks are handed to workers of the correct type.
  - Selection of virtual machine based on the current load: to this end, a load balancer was added. The current load is measured in the number of queued tasks on a VM. Each VM can run a single worker. The decision which VM handles the next task is based on the task and worker type.

To keep CloudSim as original as possible we only added features by inheritance. This makes it possible to easily update the CloudSim version. The CloudSim simulator supports two configurations at this time [1]:

- Simulation based on existing job object by creating a CloudSim workload from the executed tasks of this job. The length of the individual CloudSim tasks is taken from the time it took on the real infrastructure. This ensures that simulations follow the same execution time distribution as real jobs.
- Simulation from trace information from monitoring. This can be extended to support other formatting rules. This can be used to increase the knowledge of the data model based on output of other frameworks.

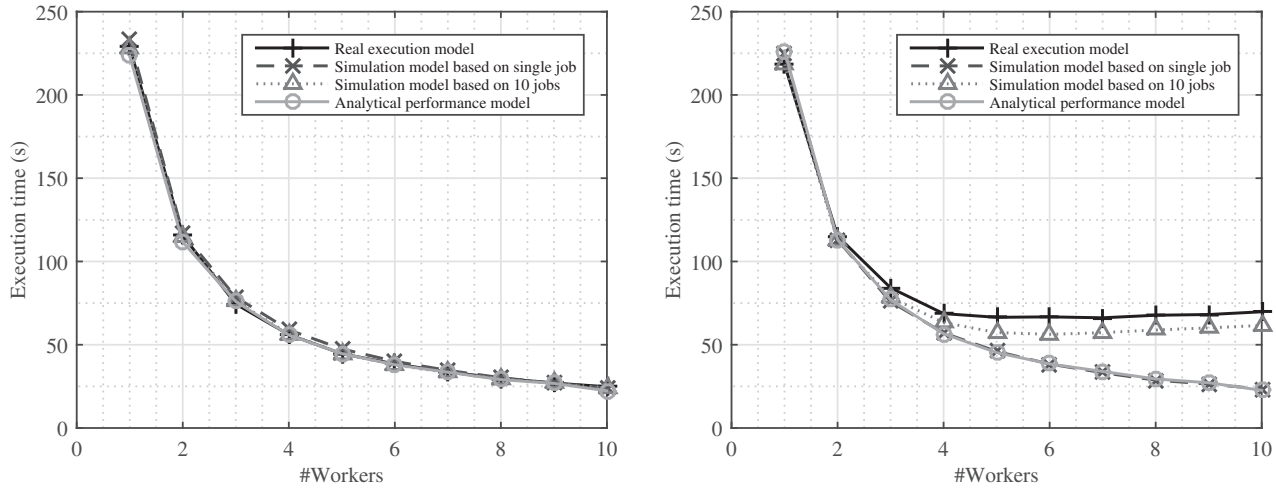
After a successful simulation events are sent to the analyzer, similar to the execution of real jobs. If the simulation has failed, the analyzer is informed so jobs will never get started with a failed simulation configuration.

5.3. Simulator evaluation

To validate the simulator results multiple bag-of-tasks jobs (see bag-of-tasks in Section 7.1), of the same job type and with 100 tasks each, were submitted on two different infrastructures. The first infrastructure is based on Openstack (2014) with four compute nodes with twelve cores each, where two cores per VM are allocated. The framework is bootstrapped to a single VM and each worker is spawned on its own VM. In our experiments, we use an infrastructure hosting at most 24 virtual machines. The second system consists of a single quad core virtual machine, where the cores are shared between all allocated worker processes. The entire framework and all workers are run on this single machine.

In Fig. 5 we compare knowledge models acquired by simulations to the real execution model acquired only from monitoring information. The simplified models are represented as a 2D graph were each point is the expected or measured wall-clock time of

[1] The customized CloudSim version can be obtained on demand. Please send an e-mail to elias.deconinck@intec.ugent.be.



(a) two core virtual machines for each worker. (b) quad core virtual machine for all workers.

Fig. 5. Comparing the real execution model of bag-of-tasks jobs based on monitoring information to two models. One from simulations based on a single job execution with one worker and the other based on 10 executed jobs on multiple worker configurations. An analytical performance model is plotted to compare against the others.

a job with the corresponding worker configuration. The 'real execution model' is the model which is calculated from monitoring information and shows the average wall-clock time for the current job type on 1–10 workers. The first simulation model is based on simulations after executing a single job with one worker and the second after executing 10 jobs with multiple workers. To compare the result we use an 'analytical performance model' based on the execution time distribution of a single task type on an Openstack VM. We can calculate the optimal execution time of a job with the formulas in (5). The analytical model closely matches the real execution model of Openstack because our infrastructure has a high throughput low latency network which does not add a lot of overhead. The average distance between those two is 1550 ms with a standard deviation of 1775 ms.

$$x = \begin{cases} 0 & (\#tasks \bmod \#workers) = 0 \\ 1 & otherwise \end{cases}$$

$$job\ execution\ time = \mu_{task} * \left(\left\lfloor \frac{\#tasks}{\#workers} \right\rfloor + x \right) \quad (5)$$

Fig. 5 (a) shows the comparison where the framework and the workers are deployed on the Openstack infrastructure. Both simulation models almost match completely with the execution model. This can be explained by the fact that our infrastructure can handle 24 VMs which do not affect each other. To further clarify the results from Fig. 5(a) we repeated the experiment on a single virtual machine where each worker has interference of all other workers resulting in the graph of Fig. 5(b). Adding more workers than threads will result in a slower wall-clock time because of the thread overhead. The simulation model based on one job is too optimistic because it has not learned the infrastructure's strength yet. After multiple simulations of 10 real submitted jobs the estimation comes closer to the real execution model. We can conclude that the simulation model adapts to the strength of the infrastructure. From Fig. 5(b) we can conclude that more than four workers is pointless on a single VM because the overhead on this infrastructure increases the execution time of the jobs. This conclusion is also visible in the **Knowledge Model** where the increasing execution time is saved for each data point. This means that the **Capacity Planner** will never select these data points because there are data points with a lower execution time and less workers.

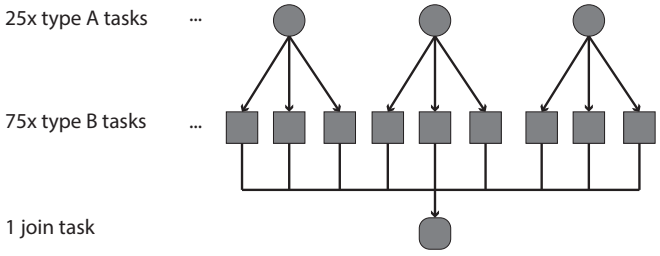


Fig. 6. Multi layered synthetic workload with multiple tasks types.

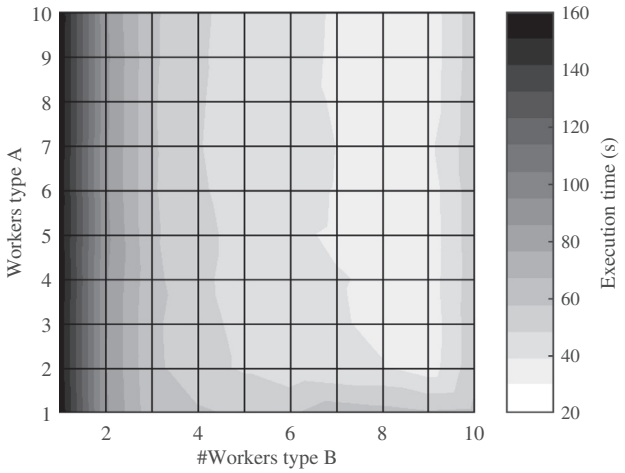
Table 3
Average and standard deviation of all tasks types of the synthetic workload from Fig. 6. Values have been acquired by running one worker for each task type on a VM with a single core.

Task type	μ (ms)	σ (ms)	#
A	2535	252	25
B	2421	238	75
Join	3	1	1

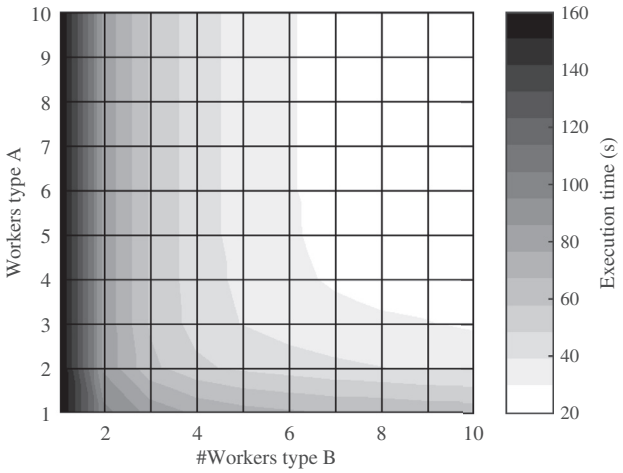
Based on the two figures from Fig. 5 we can conclude that our simulations are on point and can be used as extra data points in the knowledge model. If workers influence each other the model starts to optimistically but fixes itself by simulating extra data points from more monitoring information.

In the next experiment we evaluate the simulations on the Openstack infrastructure but with a different job type. Jobs consist of multiple layers (see Fig. 6 and Table 3) with three task types. Each worker type is increased from 1 → 10 to simulate all unique worker configuration combinations creating a three dimensional graph of the execution time and worker combinations. The 'Join' task type is excluded from the graph's axes because adding more workers will not effect the outcome. We test if the job structure can be deduced from the output of the simulations. By looking at the real execution time shown in Fig. 5.3(a) we can only see that worker type B influences the execution time drastically. Worker type A has a lot less impact.

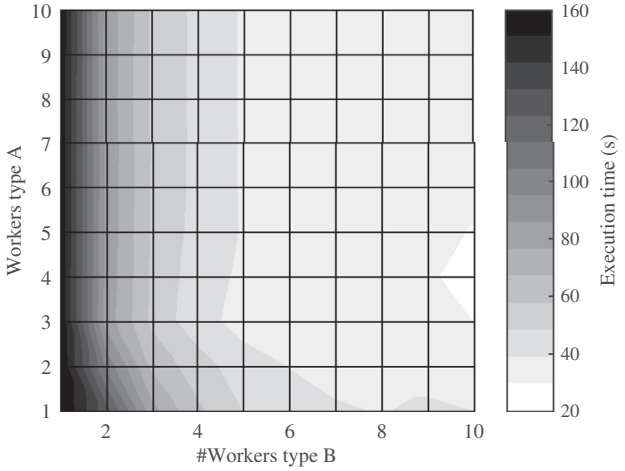
With the prediction in Fig. 5.3(b) starts from a single multi layer job executed on the Openstack infrastructure and shows already a good approximation of Fig. 5.3(a). But Fig. 5.3(c) shows



(a) Real multi layered job execution on the Openstack infrastructure.



(b) Prediction based on a multi layered job with a single worker configuration with a σ_{est} of 12 395 ms.



(c) Prediction based on multi layered jobs with all executed worker combinations from Figure 7(a). Prediction with a σ_{est} of 10 309 ms compared to Figure 7(a).

Fig. 7. 2D colormap representations of a 3D surface based on the execution time in function of number of workers for each worker type from multi layered jobs (see Section 3.3.2). The standard error of the estimates σ_{est} is the square root of the average squared error of prediction. This is used as a measure of the accuracy of the predictions.

that our simulations can represent the real execution time and learn the job structure. We see that workers of type A do not influence the execution time a lot but when there are too few workers of type A it negatively affects the execution time for more workers of type B. Because we know the structure of the multi layered job we can say that the ideal ratio of type A workers to type B workers is 1/3.

6. Design and implementation

The framework is designed as an AIOLOS Bohez et al. (2014); Verbelen et al. (2011) application. AIOLOS makes it possible to run OSGi OSGi Alliance (2003) components on any supported Operating System and creates AIOLOS nodes, which are virtual machines, containers or native processes, bootstrapped with a Java virtual machine and a OSGi runtime as requested. We used this middleware for easier scaling and reducing the necessary networking code. AIOLOS provides the possibility to design communicating components independently of their location.

AIOLOS is used to bootstrap our application on the cloud or on the local machine. It is used to boot and manage new nodes and scale Taskworker OSGi components. The private and public infrastructure providers are supported by AIOLOS and can be extended in the future. The middleware provides us with a management interface for new and existing nodes and a component interface to transparently distribute and manage OSGi components.

Our framework is split into two or more AIOLOS nodes. The first node is responsible for managing Taskworker. This includes all major OSGi bundles from AIOLOS and Taskworker. The second and all other extra nodes are worker nodes and should only have worker type bundles and optionally a storage bundle. Worker nodes are automatically scaled to accommodate for the required resources.

7. Experiments

Multiple use cases can be developed on top of our framework as an application with one or more job types. Each application needs to have at least one decider component (see Section 3) for

Table 4
Summary of all parameters for the scheduler evaluation for bag-of-task jobs. Long jobs are jobs with deadlines further away from the arrival time.

Parameters	Long jobs	Short jobs
#Tasks/job	10 tasks	
Task upper-bound param.	750, 000, 000	
Min. job deadline	35s	
Max. job deadline	70s	
#Jobs	33 jobs	67 jobs
Deadline normal distribution	$\mu = 60s; \sigma = 5s$	$\mu = 45s; \sigma = 5s$
Average arrival times	$300s \rightarrow 25s$	$150s \rightarrow 12.5s$

Table 5
Bag-of-task jobs run on the Openstack cloud. Comparing schedulers and scheduler settings to evaluate job completion before deadline. Light gray cells represent experiments which have reached the maximum capacity of the infrastructure which means the schedulers would spawn more workers if more capacity was available achieving more deadlines.

Lambda		Scheduler (#jobs completed within deadline) [%]				
Small	Long	EDF T3	EDF T2	EDF T1	EDF RL	MAX
1/150000	1/300000	34	70	100	94	100
1/100000	1/200000	26	72	100	94	100
1/50000	1/100000	32	80	90	89	100
1/25000	1/50000	39	50	88	77	100
1/12500	1/25000	8	8	11	8	12

each job type and the necessary custom workers for each task type.

First we evaluate the framework and compare schedulers using a theoretical use case with a simple job structure. Then we evaluate the acquired knowledge model based on traces from the two actual use cases introduced in Section 2. We compare the knowledge models with the analytical models calculated from the tasks durations and the jobs structures.

7.1. Theoretical workload

Bag-of-tasks job with 10 tasks where each task calculates the total number of primes until a given upper-bound (750, 000, 000) using the Sieve of Eratosthenes Weisstein (2015) algorithm. Executing these tasks on cloud virtual machines with 2 cores gives an average of 15.6 s with a standard deviation of 1.1 s.

To evaluate the planned resources and the schedulers we executed 100 bag-of-task jobs on the cloud with time-varying arrival rates. We have two Poisson processes for longer and shorter (lambda twice as large) arrival times. In Table 5 the first two columns show the varying lambda used for long and small jobs. The average arrival time for long jobs varies between 300 and 25 s and for short jobs between 150 and 12.5 s.

The job's deadlines are also divided into shorter and longer jobs to accommodate for the different Poisson arrival processes. Short jobs have a normal distribution with an average of 45 s and a standard deviation of 5 s. The longer jobs have a average deadline of 60 s and the same standard deviation. To avoid statistical outliers and keep deadlines realistically we truncated the normal distribution with a minimal and maximum deadline requirement of 35 s and 70 s. Table 4 summarizes all parameters.

We compare two scheduling algorithms and for earliest deadline first we compare 4 capacity planner settings.

EDF: Earliest Deadline First algorithm with varying capacity planning threshold. T_x defines the threshold of the capacity planner. T_1 will start an extra worker for every task waiting in the task queue, T_2 will start an extra worker when 2 tasks are waiting and T_3 for every 3 tasks waiting. Max is a special case and will run the maximum allowed workers on the cloud (for comparison reasons).

EDF RL: Rate Limited Earliest Deadline first has only one capacity planning setting and works only with the knowledge model proposed in this paper. It will calculate the resources based on the acquired knowledge.

Table 6
Average and variance VM usage for all tested schedulers on the bag-of-task jobs run on the Openstack cloud. Comparing the schedulers with lambda setting 1/100e3 to evaluate used resources to complete jobs.

Scheduler	VM Usage μ	VM Usage σ^2
EDF T3	3.77	9.79
EDF T2	3.91	17.30
EDF T1	4.97	46.15
EDF RL	4.61	30.58
MAX	20	0

Table 5 shows that for very small arrival times the number of jobs that complete before their deadline drops enormously. This happens because we limited our cloud to allow a maximum of 20 worker nodes. This is why we added MAX, which is a scheduler that instantiates the maximum allowed number of workers, to see the maximum possible number of completed jobs for 20 workers.

For offloading peak demand to other clouds we compare virtual machine usage distribution during execution of a job. This table can be viewed in Table 6. The rate limited scheduler typically has a lower variance while maintaining an acceptable amount of jobs completed before their deadline. A lower variance means less fluctuations between the number of used resources, which makes it possible to optimize the private cloud, but when needed the scheduler asks more resources which can be accommodated by a public cloud. This results in a reduced amount of jobs sent to other clouds to limit the cost.

7.2. Geographic information systems (GIS)

Geographic information system jobs consist of many fusion tasks which have three possible solutions determined by the task's properties as described in Section 2.1. The dependencies between tasks limits the maximum number of fusion workers usable for a job because each task splits into a maximum of four other tasks. We can view a GIS job as a k-ary tree with height h and root tile height 0. The upper bound for the maximum number of leaves in such a k-ary tree is k^h and the total number of tiles is $\lfloor \frac{k^{h+1}-1}{k-1} \rfloor$. The highest level of parallelism is reached when the leaf tiles are being processed so this determines the maximum usable parallel workers. The maximum usable parallel workers in our application is $4^{h_{max}}$ even if the total number of tasks for these jobs is $\lfloor \frac{4^{h_{max}+1}-1}{4-1} \rfloor$. For this experiment we have a maximum height of 4 which brings the maximum number of tasks to 341.

$$p = \frac{1}{h_{max} - h_{tile} + 1} \qquad with\ p \in \mathbb{R}]0, 1] \tag{6}$$

$$leaf\ tile = \begin{cases} true, & p \\ false, & 1 - p \end{cases}$$

This experiment focusses on a varying number of tasks and depends on the GIS application. In our GIS application each time a tile task is processed the chance of it being a leaf tile is measured with the formulas in (6). The chance p that the tile is a leaf increases the closer it gets to the maximum allowed depth and is 100% at the maximum depth. The root tile has a 20% chance of being a leaf tile. This means that these GIS jobs can have a single

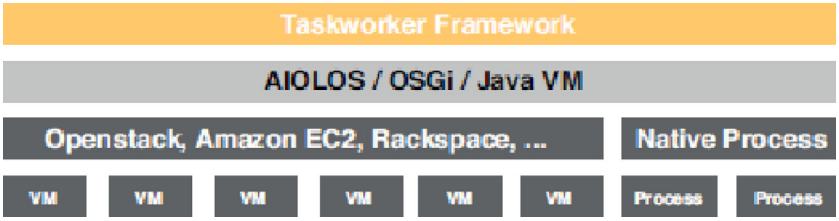


Fig. 8. Framework implementation method with AIOLOS middleware. AIOLOS supports many private and public cloud infrastructures and local testing with native processes.

Table 7
GIS application average task type length and standard deviation. The GIS application has only a single task type but the depth is not constant. A fusion task splits into four more fusion tasks or is rendered as a leaf task. These values are used to create a theoretical minimal model.

Task type	μ (ms)	σ (ms)	Average #Tasks/Job
Fusion	832	384	62

Table 8
Partial analytical model for GIS application using task characteristics in Table 7.

	Analytical model	
#Fusion workers	Execution time [s]	Ratio [%]
1	52.4	100
2	26.6	50.97
3	18.3	34.92
4	14.1	26.98
–	–	–
24	4.2	7.94
–	–	–
32	4.2	7.94
–	–	–
42	3.3	6.35

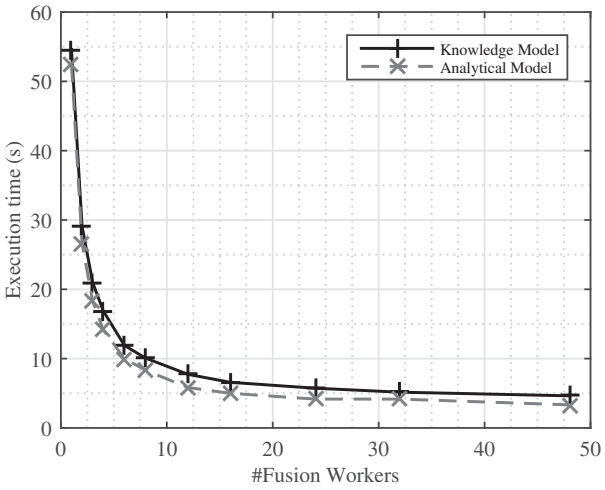


Fig. 9. GIS application knowledge model representation based on the execution time in function of the number of fusion workers (see Section 2.1). Bottom graph shows the analytical average duration of jobs based on the average values in Table 7. The real knowledge model is the average result at the end of scheduling all jobs.

task upto 341 tasks. Table 7 shows the average duration and standard deviation of fusion tasks and the average number of tasks a single job has.

Based on the values in Table 7 we can create an averaged analytical model shown in Table 8, which should be a lower limit for our knowledge model. For 63 tasks the maximum parallelism is reached with 42 workers because at tree height 3 we have a theoretical maximum of 64 leaf tiles but we only have $63 - 21 = 42$ left. $\lfloor \frac{4^{2+1}-1}{4-1} \rfloor = 21$ is the amount of tiles already processed in the levels above. From 21 to 41 fusion workers there is no improvement in the execution time because $\lceil 42/x \rceil = 2$ where $x \in \mathbb{N}[21, 41]$.

To conduct this experiment 1000 GIS jobs were submitted to the framework with an average deadline of 70 s with a standard error of 15 s. The deadline was truncated with a minimum of 25 s and a maximum of 100 s to keep the deadlines realistic. Using our own EDF RL scheduler 97 % of all deadlines were met. In the end we collected the knowledge model to compare it with the analytical model. The results are shown in Fig. 9 and shows that our model closely resembles the analytical model.

7.3. Invoice generation application

A second use case for which we have traces available is the Invoice generation application use case, introduced in Section 2.2. In contrast to the GIS application, this application focuses on multiple tenants contending resources for their similar jobs. This use case is characterized by multiple tenants that submit jobs of the same type with a different amount of parallel tasks.

We tested with 2 tenants which each have a number of invoices to process. Tenant A submits jobs with 2000 invoices while tenant B submits jobs with 2600 invoices. It should be clear that the jobs of tenant B takes longer to complete. The used deadline distribution is the same for both tenants and is truncated between 25 s and 60 s with an average execution time of 45 s and a standard error of 5 s. Each experiment submits 1000 jobs to the framework.

Running the tenants separately yields a success rate of 100% for tenant A and 99% for tenant B using the EDF RL algorithm. Fig. 10 shows how the execution times of the tenants are distributed and differ in average execution time. Submitting both tenants jobs to the same runtime creates an combined knowledge model based on all jobs and simulations. The figure shows these three distributions for a single worker combination using one worker for the types csv, zip and archive and two workers for the types template and fop. When both tenants submit to the same runtime the success rate drops slightly to 96% due to resource contention between both tenants.

Like we did for experiments in Section 7.2 we can compare the knowledge model with an analytical model based on the tasks durations. Table 9 summarizes all task types with their average execution time, standard error and the average number of tasks for each task type per job. This job type is much simpler and only scales horizontally in two task types (template, fop). Based on these values we can calculate the minimal duration of a single job as the sum of the averages for each task type. We can accomplish this minimal duration with 12 workers for the template and fop type. The analytical model can be described with equation (7). The maximum parallelism is determined by the amount of fop workers we can use, which depends on the number of tasks for fop or

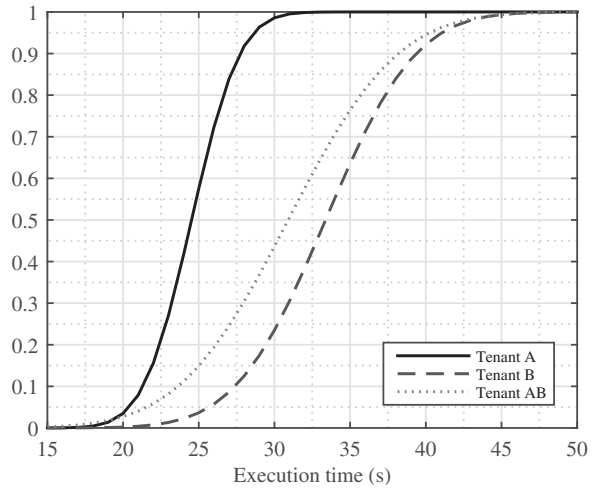


Fig. 10. Three partial invoice generation knowledge models of a single worker combination. Compare the knowledge model distribution for single Tenant A and B and a combined model AB. These models are for the worker combination: 1x csv, 2x template, 2x fop, 1x zip and 1x archive.

Table 9
Invoice generation task type parameters used to create a theoretical knowledge model. The structure of the jobs is explained in Section 2.2. The number of tasks is an average of job submission by 2 tenants with different job parameters.

Task type	μ (ms)	σ (ms)	Average #Tasks/Job
csv	51	92	1
Template	494	133	12
Fop	2603	1027	12
Zip	8100	3721	1
Archive	4	1	1

template and the lowest number of workers for these two worker types.

$$\begin{aligned} \textit{Execution time} &= 51 + 494 \\ &\quad + 1027 * \left\lceil \frac{11.5}{min(workers_{fop}, workers_{template}, 11.5)} \right\rceil \\ &\quad + 8100 + 4 \end{aligned} \tag{7}$$

Table 10 compares the analytical model based on (7) to the knowledge model acquired from submitting 1000 jobs for tenant A and B and additional simulation points. Only a small portion with the most interesting worker combinations is shown of the complete model. The ratio is the percentage compared to the execution of all tasks in sequence. A ratio of 100% means no improvement is made while a 60% value indicates a gain of 40% in execution time. The best combination is highlighted and represents the combination which performs the best with the least workers. The maximum theoretical gain for this job type is 71.79% compared to executing all tasks in sequence (first row). By comparing the ratio value of the knowledge model with the ratio value of the analytical model we can conclude that it is a good approximation. The knowledge model's execution time is within range of the analytical model's execution ± the model's error value. Some worker combinations have the exact same value and an error of '/' which means that this point is only simulated once for the same original job. Simulation points are added and selected based on the worker requirements of previous jobs so if more jobs with stricter deadlines are submitted these inaccurate worker combinations would be simulated again and get a new error value.

Table 10
Partial knowledge model key value map from the invoice generation job type for 2 tenants (AB) with different job parameters. The highlighted row is the best possible solution for this job type, adding more workers would result in the same value.

#Workers Type csv,template, fop,zip,archive	Knowledge Model Execution time [s]	Error [s]	Ratio [%]	Analytical Model Execution time [s]	Ratio [%]
1,1,1,1,1	47.4	17.5	100	39.9	100
1,1,2,1,1	31.5	8.1	66.39	39.9	100
1,1,3,1,1	28.6	3.6	60.44	39.9	100
1,2,1,1,1	44.9	5.3	94.72	39.9	100
1,2,2,1,1	30.9	5.7	65.21	24.3	60.84
1,2,3,1,1	25.1	4.1	53.00	24.3	60.84
1,3,2,1,1	31.0	5.5	65.30	24.3	60.84
1,3,3,1,1	24.9	3.8	52.60	19.1	47.79
..	..	..	..	..	..
1,6,6,1,1	17.3	/	36.48	13.9	34.74
1,6,12,1,1	13.3	1.9	28.00	13.9	34.74
..	..	..	..	..	..
1,12,12,1,1	12.0	/	25.33	11.2	28.21
1,24,24,1,1	12.0	/	25.33	11.2	28.21
..	..	..	..	..	..
1,1,1,1,2	41.0	5.6	86.53	39.9	100
1,1,1,2,1	46.8	12.6	98.64	39.9	100
2,1,1,1,1	54.2	17.6	114.24	39.9	100

8. Related work

Provisioning resources and service workload scheduling are known as classical NP-complete problems (Wieczorek et al., 2008; Yu et al., 2008). This means that generating an optimal solution is impossible in polynomial time so algorithms focus on generating a sub-optimal solution which come close to the optimal. Scheduling algorithms have been studied intensively and are the focus of most papers. Dynamic resource provisioning in a hybrid cloud for service workloads while taking care of the users Quality of Service (QoS) expectations, like deadline and budget, are equally important but less researched on.

Some surveys Xavier and Lovesum (2013); Pooja and Kumari (2013) have been submitted to evaluate different scheduling algorithms based on QoS constraints. Because these have been extensively tested and researched we can use the information to adapt our own algorithm in our framework. The focus of this paper is not creating and testing a new scheduling algorithm but using and adapting a proven algorithm. We decided to use an easy implementable algorithm to test our system.

In previous work Deboosere et al. (2012); Vankeirsbilck et al. (2014), we have designed scheduling algorithms that maximize QoS with a given set of computing resources. Specifically, we employed resource overbooking strategies that take heterogeneity of compute and I/O blocking tasks into account to further exploit parallellism with statistical prediction of resource contention. Furthermore, we introduced algorithms to consolidation of VMs onto a minimum of machines to enable shutting down redundant ones to minimize server energy consumption. Although the main goal of the current work is to spill over to efficiently manage additional cloud resources if necessary, these concepts are perfectly combinable with the algorithm presented in the current manuscript.

In Mao et al. (2010), each submitted job is scheduled in FCFS order and is not optimized for cost or deadline. The main goal is to auto-scale the infrastructure to reach the jobs deadline within the budget. This approach does not take into account scheduling multiple jobs on a dynamic infrastructure with the minimum resources. The same conclusions can be drawn for Mao and

Humphrey (2011) which uses EDF task scheduling instead of FCFS. Both papers assume the structure of workloads and the average duration is known. The services needed to execute the tasks are auto-scaled to the cloud. In our system AIOLOS manages the workers to execute each task and can rapidly scale-up or down based on the needs of all jobs currently running. This means that the optimization is done on the entire workload of all jobs.

In den Bossche et al. (2010) outsourcing tasks from a private cloud or data center to a public cloud in times of heavy load is discussed. Since public cloud providers have been more accessible to a wider range of consumers the outsourcing process has got more attention. Outsourcing tasks wile maximizing the internal data center utilization and minimizing the cost while meeting the QoS constraints is a hard task. The difference between there approach and ours is the fact that we do not assume we have knowledge about the makespan of jobs. They know for each virtual machine type the runtime of each task. As a second they assume that each application or workload consists of a number of trivially parallel or independent tasks. They do not offer functionality for service workloads.

A market-oriented approach is discussed in Wu et al. (2011). They discuss a solution for scheduling service workloads in a Hybrid cloud taking into account QoS constraints. They propose a hierarchical scheduling strategy in a cloud workflow system. They use the same basic requirements that are stated in this paper: (1) satisfaction of QoS requirements for job instances; (2) minimisation of the public cloud running cost; (3) good scalability for optimising Task-to-VM assignment in public and private clouds. The main focus of the paper is to optimize the scheduling for multiple workloads in a hybrid limited cloud. Our main focus is to estimate en plan unknown workloads on a hybrid cloud with unlimited capacity.

The last paper Rodriguez and Buyya (2014) proposes a resource provisioning and scheduling strategy for scientific workflows (DAG) on IaaS clouds. They present an algorithm based on meta-heuristic optimization technique, particle swarm optimization (PSO), which aims to minimize the overall workflow makespan while meeting deadline constraints. The algorithm is evaluated using CloudSim. They focus on IaaS clouds which offer unlimited heterogeneous resources that can be accessed on demand. They incorporate the same two stages as us: (1) resource provisioning; (2) scheduling task to the best-suited resource. Unlike us they assume task runtime is known beforehand (in FLOPS). We learn from previously executed jobs the structure and makespan with varying number of workers to improve the frameworks knowledge. This makes sure that the estimated execution time is based on the strength of a virtual machine and the actual execution.

Most papers use some sort of cloud simulator to evaluate their algorithm and/or framework. We try to use the simulations as input to our framework to further improve the knowledge about workload structure and makespan. By closing the loop we can make better estimations of newly submitted jobs.

9. Conclusion and future work

Deadline-constrained task scheduling for jobs with an uncertain, dynamically changeable task structure is challenging. In this paper, we have presented a model-driven approach, where the main characteristics of the tasks in the job structure are learned over time. The model estimates the job makespan as a function of the number of resources applied to task workers. The resulting knowledge model allows for efficient planning and scaling of resources and is particulary useful for cloud providers as it allows them to assess the infrastructure needed to execute a deadline-constrained workload without knowing the internal business logic of the workload. We have demonstrated the benefits with two use cases, one with a dynamic job graph depth and the other with varying graph width.

The model is improved with each execution of the workload on the cloud infrastructure. For cost reasons, it is however impossible to run the workload with every possible combination of resources. Instead, we have demonstrated that CloudSim simulations can be used to expand the knowledge model with more configurations.

We compared analytical models of well known job structures to knowledge models acquired by simulating previous jobs. The workloads can gradually change over time and the estimation model will adapt to accommodate this change. The accumulated knowledge can reproduce the real execution of a workflow. This knowledge can be used as a guideline for dynamic scheduling algorithms like rate limited earliest deadline first. The experiments show that this scheduler can complete most jobs before their deadline without using excessive resources. This scheduler does not give an optimal solution but finds a good approximation from the knowledge acquired from previous runs.

In the future we will improve the scheduler and adapt the simulator so it takes into account more constraints. These extra constraints are different virtual machine types, data locality, security and cost constraints of public clouds. Virtual machines on public clouds are paid per hour and this property is currently not used in our calculations. The main goal is to reduce the cost of overspilling to the public cloud and estimate the optimal size and computing power of the private cloud given the history of the hybrid cloud usage.

Acknowledgments

This research is partially funded by the iMinds project DREAMaaS. Steven Bohez is funded by Ph.D. grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

References

Bohez, S., De Coninck, E., Verbelen, T., Simoens, P., Dhoedt, B., 2014. Enabling component-based mobile cloud computing with the aiolos middleware. In: Proceedings of the 13th workshop on adaptive and reflective middleware. ACM, New York, NY, USA, pp. 2:1–2:6. doi:10.1145/2677017.2677019.

den Bossche, R.V., Vanmechelen, K., Broeckhove, J., 2010. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In: Proceedings of the 2010 IEEE 3rd international conference on cloud computing, pp. 228–235. doi:10.1109/CLOUD.2010.58.

Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R., 2011. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw: Pract Exp 41 (1), 23–50. doi:10.1002/spe.995.

Deboosere, L., Vankeirsbilck, B., Simoens, P., Turck, F., Dhoedt, B., Demeester, P., 2012. Efficient resource management for virtual desktop cloud computing. J Supercomput 62 (2), 741–767. doi:10.1007/s11227-012-0747-0.

Mao, M., Humphrey, M., 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. ACM, New York, NY, USA, pp. 49:1–49:12. doi:10.1145/2063384.2063449.

Mao, M., Li, J., Humphrey, M., 2010. Cloud auto-scaling with deadline and budget constraints. In: 2010 11th IEEE/ACM international conference on grid computing, pp. 41–48. doi:10.1109/GRID.2010.5697966.

OSGi Alliance, 2003. Osgi service platform, release 3. IOS Press, Inc.

Pooja, Kumari, N., 2013. Performance evaluation of cost-time based workflow scheduling algorithms in cloud computing. Int J Adv Res Comput Sci Softw Eng 3 (9), 437–439.

Rodriguez, M.A., Buyya, R., 2014. Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds. IEEE Trans Cloud Comput 2 (2), 222–235. doi:10.1109/TCC.2014.2314655.

Sotomayor, B., Montero, R.S., Llorente, I.M., Foster, I., 2009. Virtual infrastructure management in private and hybrid clouds. IEEE Internet Comput 13 (5), 14–22. doi:10.1109/MIC.2009.119.

Vankeirsbilck, B., Deboosere, L., Simoens, P., Demeester, P., Turck, F., Dhoedt, B., 2014. User subscription-based resource management for desktop-as-a-service platforms. J Supercomput 69 (1), 412–428. doi:10.1007/s11227-014-1171-4.

Verbelen, T., Simoens, P., De Turck, F., Dhoedt, B., 2011. Aiolos: Mobile middleware for adaptive offloading. In: Proceedings of the workshop on posters and demos track. ACM, New York, NY, USA, pp. 20:1–20:2. doi:10.1145/2088960.2088980.

Weisstein, E. W., 2015. Sieve of eratosthenes. From MathWorld–A Wolfram Web Resource. @Online, http://mathworld.wolfram.com/SieveofEratosthenes.html.

Wieczorek, M., Hoheisel, A., Prodan, R., 2008. Grid middleware and services: challenges and solutions. In: Taxonomies of the multi-criteria grid workflow scheduling problem. Springer US, Boston, MA, pp. 237–264. doi:10.1007/978-0-387-78446-5_16.

Wu, Z., Liu, X., Ni, Z., Yuan, D., Yang, Y., 2011. A market-oriented hierarchical scheduling strategy in cloud workflow systems. J Supercomput 63 (1), 256–293. doi:10.1007/s11227-011-0578-4.

Xavier, S., Lovesum, S.J., 2013. A survey of various workflow scheduling algorithms in cloud environment. Int J Sci Res Publ 3 (2), 1–3.

Yu, J., Buyya, R., Ramamohanarao, K., 2008. Metaheuristics for scheduling in distributed computing environments. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 173–214. doi:10.1007/978-3-540-69277-5_7.

2014. Amazon simple workflow service (swf) @Online, http://aws.amazon.com/swf/

Openstack 2014. Open source software for building private and public clouds. @Online. http://www.openstack.org/.

Elias De Coninck received his M.Sc. in Information Engineering Technology from University College Ghent, Belgium in August 2012. He is working on a Ph.D. at Ghent University - iMinds on algorithms for optimally scaling application components in hybrid cloud systems. He is mainly responsible for running AIOLOS in the cloud and the jclouds integration.

Tim Verbelen received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2009. In July 2013, he received his Ph.D. degree with his dissertation "Adaptive Offloading and Configuration of Resource Intensive Mobile Applications". Since August 2009, he has been working at the Department of Information Technology (INTEC) of the Faculty of Engineering at Ghent University, and is now active as postdoctoral researcher. His main research interests include mobile cloud computing and adaptive software. Specifically he is researching adaptive strategies to enhance real-time applications such as Augmented Reality on mobile devices.

Bert Vankeirsbilck received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2007. In June 2013, he received his Ph.D. degree with his dissertation "Optimisation of Quality of Experience for Mobile Thin Client Systems" He has been working at the Department of Information Technology (INTEC) of the Faculty of Engineering at Ghent University, and is now active as postdoctoral researcher.

Steven Bohez received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2013. He is working on a Ph.D. at Ghent University - iMinds and is focusing on advanced mobile cloud applications that are distributed between mobile devices and the cloud. He is currently researching optimal deployment and scaling algorithms.

Pieter Simoens received his M.Sc. degree in Electrotechnical Engineering in July 2005 from Ghent University. In February 2011, he obtained a Ph.D. degree with his dissertation "Thin Client Protocol Optimizations for Mobile Cloud Computing". Since October 2005, he has been working at the Department of Information Technology (INTEC) of Ghent University. During his Ph.D., he has been active as researcher in the FP6 MUSE project and as work package leader in the FP7 MobiThin project. He is now active as post-doctoral researcher, focusing on smart clients and cloud computing. The research activities are combined with a mandate as doctoral assistant at the Hogeschool Gent (HoGent).

Bart Dhoedt received a Masters degree in Electro-technical Engineering (1990) from Ghent University. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a Ph.D. degree in 1995. After a 2-year post-doc in opto-electronics, he became Professor at the Department of Information Technology. Bart Dhoedt is responsible for various courses on algorithms, advanced programming, software development and distributed systems. His research interests include software engineering, distributed systems, mobile and ubiquitous computing, smart clients, middleware, cloud computing and autonomic systems. He is author or co-author of more than 300 publications in international journals or conference proceedings.