Cost-aware Scheduling of Software Processes Execution in the Cloud

Sami Alajrami[1], Alexander Romanovsky[1] and Barbara Gallina[2]
[1]*School of Computing, Newcastle University, Newcastle upon Tyne, U.K.*
[2]*Mälardalen University, Västerås, Sweden*

Keywords: Dynamic Scheduling, Software Processes, Cloud Computing, Cloud-based Software Process Execution.

Abstract: Using cloud computing to execute software processes brings several benefits to software development. In a previous work, we proposed a reference architecture, which treats software processes as workflows and uses cloud computing to execute them. Scheduling the execution in the cloud impacts the execution cost and the cloud resources utilization. Existing workflow scheduling algorithms target business and scientific (data-driven) workflows, but not software processes workflows. In this paper, we adapt three scheduling algorithms for our architecture and propose a fourth one; the Proportional Adaptive Task Schedule algorithm. We evaluate the algorithms in terms of their execution cost, makespan and cloud resource utilization. Our results show that our proposed algorithm saves between 19.74% and 45.78% of the execution cost and provides the best resource (virtual machine) utilization compared to the adapted algorithms while providing the second best makespan.

1 INTRODUCTION

Cloud computing has become the delivery platform for the Post-PC era applications (Alajrami et al., 2016b). In a previous work (Alajrami et al., 2016b), we argued that software development processes can also benefit from the cloud virtues and be delivered as a service.

Paulk et al. (Paulk et al., 1993) describe a software process as "a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals)". Software processes are processes too (Fuggetta, 2000) and the use of process-related technologies (e.g., workflow systems) for software processes has been overlooked (Fuggetta and Di Nitto, 2014).

We proposed a reference architecture for Software Development as a Service (SDaaS) (Alajrami et al., 2016b) where software development processes are modeled then executed in the cloud. The architecture is depicted in Figure 1 and we briefly describe it in Section 2.

Software vs Business Processes

Software processes differ from business and scientific (data-driven) processes. A business process automates a business procedure which has been well-defined in a given context. Defining such a process is often mainly concerned with the business side of things. For example, the process of admitting new students into a university is a business process which can be modelled and executed in a BPM system. On the other hand, building a software that can execute the student admission procedure requires considering more aspects than just the business side. Those aspects include: security, performance, software/hardware resources, legislation and compliance etc. As a result, the software development process of such system would go through different stages and will include business, technical and legal stakeholders. The process itself will have several sub-processes for the specifications, design, implementation, quality assurance, testing, releasing and operating the software. All these (sub)processes are subject to frequent change during the project (e.g., when the business requirements change). In short, software process are more dynamic and interactive compared to business processes.

Software vs Scientific Processes

The execution and scheduling of scientific processes have been addressed in other works. However, as we explained earlier, software processes are more dynamic and interactive. A scientific process is often data-driven (sequential/parallel processing of data to obtain some results) which makes scheduling its execution fundamentally different from scheduling a software process.

Alajrami, S., Romanovsky, A. and Gallina, B.
Cost-aware Scheduling of Software Processes Execution in the Cloud.
DOI: 10.5220/0006607902030212
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 203-212
ISBN: 978-989-758-283-7

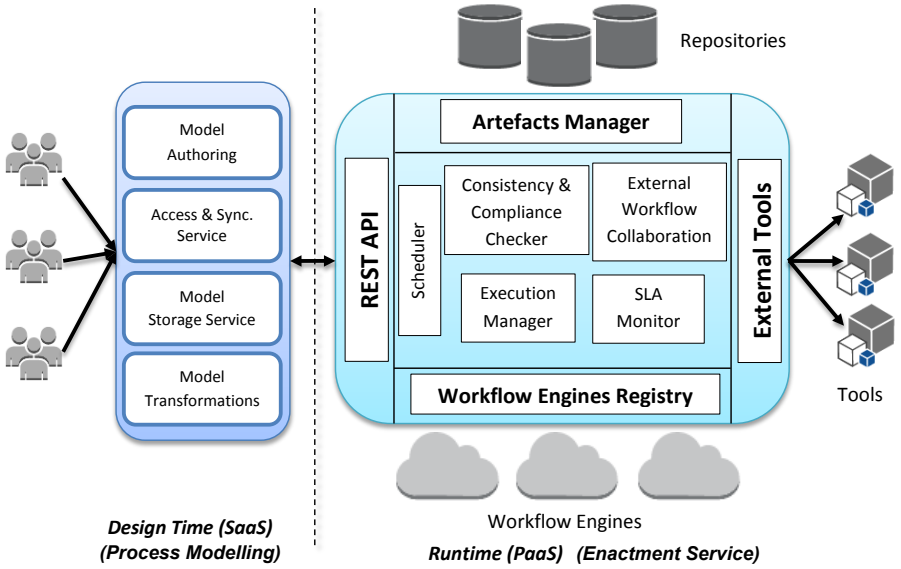

Figure 1: The SDaaS reference architecture. Taken from (Alajrami et al., 2016b).

Several approaches to software process modeling have been introduced over time. The state-of-the-art approach is the UML-based approach which utilizes the wide adoption and acceptance of the Unified Modeling Language (UML) for modeling software processes. The most notable UML-based language is the OMG SPEM 2.0 (OMG, 2008) which contains the necessary elements to model software processes. However, as we explained in (Alajrami et al., 2016a), SPEM 2.0 lacks explicit support for process execution in general and for cloud-based execution in particular. Thus, we proposed EXE-SPEM (Alajrami et al., 2016a); an extension of SPEM 2.0, which can model cloud-based executable software processes. EXE-SPEM models describe a partially ordered set of software development activities and their required computational and tool support (e.g., some activities may require execution on private infrastructure for security reasons). These activities can either be: *concrete* activities (the tool support that is used for process execution) or *control* activities (enables guiding the execution of the process in one of multiple predefined directions).

Executing software processes in the cloud harnesses the cloud economies of scale. However, despite the illusion that the cloud offers an unlimited pool of computational resources, these resources come at a cost. While computational resources might be plenty, monetary resources are always limited. Therefore, **software processes execution scheduling should consider reducing the cost while not causing significant execution delays**. Workflow scheduling in the cloud has been investigated (e.g., (Vijindra and Shenai, 2012; Singh and Singh, 2013; Bala and Chana, 2011)) where several algorithms have been proposed with different objectives (cost reduction, meeting deadlines, makespan optimization, etc.). However, **these algorithms have always focused on scientific or business process workflows and none have addressed execution of software processes workflows**. Some algorithms use static scheduling mechanism, which does not handle the dynamicity and heterogeneity of cloud resources. In addition, few algorithms considered the diverse requirements that different workflows activities may require in terms of cloud resource types. To the best of our knowledge, no existing research has addressed scheduling software processes workflows in the cloud and the catering for the special needs of such workflows.

In a setting where multiple software development processes (and their activities) compete for shared computational resources (workflow engines which execute the process), scheduling software processes (workflows) execution becomes important. Workflow scheduling is an NP-hard problem (Wang et al., 2014; Yu and Buyya, 2005) which refers to the allocation of sufficient resources (human or computational) to workflow activities. The schedule impacts the workflow makespan (execution time) and cost as well as the computational resources utilization.

In this paper, we focus on the *Scheduler* component from the SDaaS architecture. To reduce the software processes execution cost in the cloud, we adapt three algorithms (which were not designed to schedule software processes execution) for scheduling software processes execution in the cloud. In addition, we

propose a fourth algorithm; the Proportional Adaptive Task Schedule. We evaluate these algorithms through simulation and we benchmark their performance in terms of execution cost and makespan. The simulation results show that our proposed algorithm saves between 19.74% and 45.78% of the execution cost and provides the best resource (virtual machine) utilization compared to the adapted algorithms while providing the second best makespan.

The rest of the paper is structured as follows: the next section provides background on workflow scheduling algorithms. Section 4 describes the adapted and proposed scheduling algorithms for software processes execution in the cloud. Section 5 demonstrates the evaluation of the adapted and proposed algorithms. Finally, section 6 provides concluding remarks.

2 SDAAS OVERVIEW

As shown in Figure 1, the SDaaS architecture consists of: design-time components for modeling and manipulating software processes, and run-time components for executing software process models and manage their artifacts.

Software processes are modeled at design-time using the EXE-SPEM (Alajrami et al., 2016a) modeling language. Then, these models are executed at runtime. The modelling components are extensively explained in (Alajrami et al., 2016b).

The *Scheduler* is responsible for allocating the execution of the software process activities to workflow engines which match the modeled (computational and privacy) requirements for each activity. The *Workflow Engines Registry* keeps track of the available workflow engines and their specifications and workloads. The *Execution Manager* triggers the execution and monitors it at run-time. During the execution of activities, *External Tools* can be used and generated artifacts are managed and stored by the *Artifacts Manager*. *External Workflow Collaboration* enables software processes to interoperate with other workflow systems. Further, when part of the process is outsourced to partner(s), the *SLA Monitor* ensures that all parties are not breaching the SLA. Finally, the *Consistency & Compliance Checker* performs consistency checking for the process (during its execution) towards a standard process. This can alleviate development problems (e.g., deviating from a standard process) early and save time and cost. The SDaaS architecture treats and executes software processes as workflows. The execution takes place in a set of distributed workflow engines (execution containers for executing workflow activities) deployed in a public, private or hybrid cloud (mixture of public and private cloud).

3 WORKFLOW SCHEDULING ALGORITHMS

While several authors have surveyed workflow scheduling algorithms (e.g., (Vijindra and Shenai, 2012; Singh and Singh, 2013; Bala and Chana, 2011)), in this section, for brevity, we review only three workflow scheduling algorithms and their suitability for scheduling software processes execution in the cloud. These algorithms were selected as they target optimizing workflow execution cost and/or makespan in the cloud.

A Compromised-Time-Cost Scheduling Algorithm in SwinDeW-C for Instance-Intensive Cost-Constrained Workflows on a Cloud Computing Platform

Overview

Liu et al. (Liu et al., 2010) proposed an algorithm for scheduling workflows with large number of instances (instance-intensive) and cost constraints on the cloud. It aims to minimize the cost under user designated deadlines or minimizing execution time under user designated budget. The algorithm dynamically calculates the relation between cost and execution time and visualizes it to the user so that he/she can make a choice to compromise time or cost. The algorithm is compared against the Deadline-MDP algorithm (Yu et al., 2005) in terms of cost and makespan and shows that it reduces execution cost by over 15% whilst meeting the user-designated deadline and reduces the mean execution time by over 20% within the user-designated execution cost.

Limitations

However, it is worth noting that the cost calculation in this algorithm does not consider the execution time taken by a task and instead uses a hard-coded table for execution prices based on the provided processing speed. In addition, this algorithm does not support tasks to have special resource requirements such as private resources or specific computational power which is needed in software processes as we explained in Section 1. Additionally, the idea of applying deadlines to software processes workflows is not practical due to the fact that it is hard to predict/-control the execution time of many activities in such processes. Especially, the ones which rely on human intervention.

Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows

Overview

Mao et al. (Mao and Humphrey, 2011) proposed an algorithm for scheduling workflow tasks within a given deadline and at the minimal cost by dynamically allocating/deallocating virtual machines (VMs). The schedule is dynamically calculated to auto-scale VMs to handle dynamic loads from multiple workflows.

Limitations

While this approach would fit for data-intensive or business process workflows, as mentioned earlier, allocating deadlines for software processes is not practical. Software processes have a mixture of human-performed and tool-supported tasks. The human-performed tasks are often unpredictable and can be long-running, therefore, it would be challenging to allocate sub-deadlines for this type of tasks.

Adaptive Task Schedule

Overview

In (Wang et al., 2014), Wang et al. proposed a dynamic adaptive task schedule algorithm which dynamically sets a maximum number of VMs that can be acquired at any given time. This limit is calculated based on two variables: either historical (backward) or future (forward) number of activities and an arbitrary threshold. They compare this algorithm with three other algorithms (one static and two dynamic) and their results show that the adaptive task schedule algorithm based on future number of tasks gives the best performance.

Limitations

This algorithm, however, does not handle specific requirements of each workflow activity (which is needed for software processes as we discussed earlier) and relies on an arbitrary value which does not have any rules to calculate.

We adapt this algorithm (in Section 4.2) to the SDaaS architecture needs and we show that our proposed algorithm (in Section 4.3) outperforms this one.

4 COST-AWARE SCHEDULING ALGORITHMS

Unlike scientific workflows, software processes are control-flow workflows which involve more human interactions. Furthermore, different types of software processes tasks can require different types of resources in terms of computational power and/or deployment choices (public vs private clouds). These requirements include the choice of public or private cloud, cloud provider (in case of using public clouds), the virtual machine image, machine type (specifying the amount of memory, CPU power and network bandwidth) and number of machines (in case of a distributed activity).

Below, we list the **assumptions** we use to design the scheduling process:

- The SDaaS architecture will be used by an organization which have multiple (geographically-distributed) teams which collaborate on several projects concurrently.
- Software processes contain a set of activities with different requirements for execution privacy and computational resources.
- Some activities may be required to be performed quickly while others may not. In a delayed project, certain activities will be required to be performed quickly to avoid further delays. In addition, critical activities that precede the execution of many other activities are naturally expected to be performed faster so that they do not block other activities longer. These activities are referred to as *priority activities*.
- Interactive activities are not executed on the cloud since these activities may involve stakeholders performing certain tasks offline. Likewise, scheduling the human activities (the ones preformed solely by humans without any tool support) is not considered since it does not have an impact on the cost of using the cloud.
- An activity becomes ready for execution once all of its input artefacts become available.
- At any given time, there might be several ready-to-execute activities from different processes.
- Activities execution times are presumed to be known. Execution time estimation techniques are available (e.g., (Jang et al., 2004)) but are out of the scope of this chapter.
- The cost of executing an activity is dependent on the time it takes to finish and the cost of data transfer outside the cloud provider boundary. For simplicity, both the public and the private cloud resources are assumed to be located within two data centres (one public and one private) and data transfer between them is negligible. Therefore, data transfer costs are assumed to be negligible.

The **objectives** of software process scheduling are:

1. To allocate activities to a workflow engines pool containing engines which match the required resources by the activity.
2. To reduce the overall workflows cloud-based execution cost by switching workflow engines on/off when needed/unneeded.

3. Reducing the cost conflicts with the workflow makespan (execution time). The scheduling should minimize the impact of reducing the cost on the workflow makespan.
4. To utilize the available workflow engines as best as possible.

4.1 Terminology

Here, we define the terms related to the scheduling process in the SDaaS reference architecture:

- Workflow engines pool: is a pool of workflow engines deployed on similar virtual machines (in terms of computational power and deployment model).
- Workflow engines pool size (R): is the maximum number of active workflow engines a pool can have at any given operational hour.
- Workflow engine operational hours: are the hourly units of time starting from the time a workflow engine starts.
- Workflow execution time: is the difference between the execution start time of the first activity in the workflow and the execution end time of the last activity in the workflow.
- Execution cost: is the cost of executing all the desired workflows in the SDaaS architecture. This can be calculated by aggregating the cost of running each workflow engine instance as follows:

$$Cost = \sum_{i=1}^{n} VM_n * t_n \quad (1)$$

Where VM_n is the price per partial hour for running the virtual machine hosting the workflow engine and t_n is the number of partial hours that the workflow engine has been running.

4.2 Adapted Algorithms

In this subsection, we explain the adaptation of three different scheduling algorithms to fit for scheduling the execution of software processes in the cloud. Two of these algorithms are adapted from the first come first serve algorithm and the third one is adapted from the Adaptive Task Schedule algorithm (see Section 3).

Unlimited First Come First Serve (UFCFS). This is the simplest and most basic scheduling algorithm where the pool size R is always set to infinity. Once an activity is ready-to-execute, it is allocated to an available workflow engine in the relevant pool (if exists),

```
Activity A;
List<WorkflowEnginePool> pools;

start
find a pool in pools which match the
    computational resources and privacy
     requirements of A.
if(pool is found)
{
    find an available workflow engine
    if(workflow engine is found)
        add A to the jobs queue of the
    engine;
    else
    {
        create and start a new workflow
     engine and add A to its jobs queue
    ;
    }
}
else
{
    create a pool;
    create and start a new workflow
    engine in the new pool and add A to
     its jobs queue;
}
end
```

Figure 2: Unlimited First Come First Serve algorithm.

otherwise a new pool and/or workflow engine are created. Figure 2 shows the pseudo code of the *UFCFS* algorithm.

Limited First Come First Serve (LFCFS). This is a similar algorithm to the UFCFS except that there is a limit on the number of active workflow engines in any pool at any time. Figure 3 shows the LFCFS algorithm. The pool size limit (R) is an arbitrary universal value which aims to restrict the execution cost. If all workflow engines in a pool are busy and their number has reached R and a new activity is ready to be executed in this pool, the scheduler will allocate this activity to the workflow engine with the earliest finishing time. This means that the activity will be delayed until a suitable workflow engine becomes available again.

Pool-based Adaptive Task Schedule. This algorithm is adapted from the Adaptive Task Schedule algorithm (Wang et al., 2014) described in Section 3. Unlike the original algorithm which has two versions (one looking forward and one backward), here we only look at the expected activities in the next hour (forward). Since the activities arrive in a non-deterministic way, the history alone does not necessarily give an accurate prediction for the predicted load in the next hour. Another difference is that we match the activities to pools and calculate R dynamically for each pool rather than for the entire system,

```
Activity A;
List<WorkflowEnginePool> pools;
int R; // the max number of workflow
    engines in any pool

start:
find a pool in pools which match the
    computational resources and privacy
     requirements of A.
if(pool is found)
{
    find an available workflow engine;
    if(workflow engine is found)
        add A to the jobs queue of the
    engine;
    else
    {
        if(number of workflow engines
    in pool i < R)
            create and start a new
    workflow engine and add A to its
    jobs queue;
        else
            allocate A to the first
    available engine;
    }
}
else
{
    create a pool;
    create and start a new workflow
    engine in the new pool and add A to
     its jobs queue;
}
end
```

Figure 3: Limited First Come First Serve algorithm.

hence the name *Pool-based*. This means that **each pool can have a different pool size calculated dynamically on every operational hour** using the following formula:

$$R_i = T * E_i \tag{2}$$

Where T is a universal arbitrary real value between 0 to 1 which indicates the proportion between the activities to be executed and the workflow engines. For example, when T is 0.5, it means that there should be a workflow engine for each two activities. E_i is the number of activities which match pool i and are expected to be ready for execution in the next hour.

Figure 4 shows this algorithm. As we can see, the algorithm is very similar to the LFCFS algorithm except that each pool has its own R.

```
Activity A;
List<WorkflowEnginePool> pools;
List<int> R; // the max number of
    workflow engines for each pool

start:
find a pool in pools which match the
    computational resources and privacy
     requirements of A.
if(pool is found)
{
    find an available workflow engine;
    if(workflow engine is found)
        add A to the jobs queue of the
    engine;
    else
    {
        if(number of workflow engines
    in pool i < Ri)
            create and start a new
    workflow engine and add A to its
    jobs queue;
        else
            allocate A to the first
    available engine;
    }
}
else
{
    create a pool;
    create and start a new workflow
    engine in the new pool and add A to
     its jobs queue;
}
end
```

Figure 4: Pool-based Adaptive task scheduling algorithm adapted from (Wang et al., 2014).

4.3 The Proportional Adaptive Task Schedule Algorithm

Similar to the Pool-based Adaptive Task Schedule and the LFCFS algorithms, this algorithm restricts the creation of new workflow engines in a pool by a maximum limit R. The difference is that R is now calculated based on the proportion between the execution time of the activities that are predicted to start in the next hour and those which have started execution in the past hour. The following formula is applied when R_i for a given pool i is calculated for the first time:

$$R_i = \left\lfloor \frac{T_{next}}{60} \right\rfloor \tag{3}$$

Where T_{next} is the total execution time of the activities that will start in the next hour (in minutes). Therefore, R_i is the floor of the expected execution hours needed to execute the activities that would start in the next hour. When R_i has been set before, the

following formula is applied to calculate R_i on every operational hour:

$$R_i = \left\lceil \frac{T_{next}}{T_{past}} * R_i' \right\rceil \quad (4)$$

Where T_{past} is the total execution time (in minutes) of the activities that have started in the past hour and R_i' is the last value of R_i. The algorithm itself is the same as the pool-based adaptive task schedule algorithm presented in Figure 4.

5 EVALUATION

To analyze the performance of the algorithms described in the previous section, we simulate the execution of each algorithm and measure two metrics: (a) the makespan for each simulated workflow, and (b) the total cost of executing all workflows. The simulation uses the four algorithms to schedule activities from multiple workflow instances. In order to simulate a real workflow execution scenario, we generate requests to execute workflow instances at random times to create non-determinism. In a real scenario, workflow instances can be requested to be executed at any time and might be executing in parallel with some other instances. Since randomization is used, there is a need to run the simulation several times and calculate mean values for the desired metrics.

5.1 Input Process Models

We use three input workflow models of sizes 7, 9 and 10 activities. These models have different requirements for activities (a mixture of public/private and priority/non-priority activities). The structure of these models and which activity has which requirements are irrelevant as the simulator randomly chooses a time to trigger the request for each of the three input models in each simulation iteration. This creates a non-deterministic load on different computational resources.

Figures 5, 6 and 7 illustrate the three input workflow models. Each activity is colour coded to identify the machine type it requires. The execution time is specified above each activity where a * symbol indicates that the activity is a priority activity.

5.2 Workflow Engines

The workflow engines are where the execution of activities takes place. For the purpose of this evaluation, we are only concerned about how long it takes to execute an activity. Workflow engines are hosted

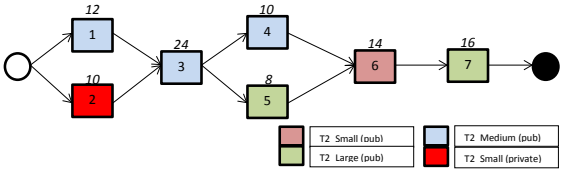

Figure 5: The first workflow input model.

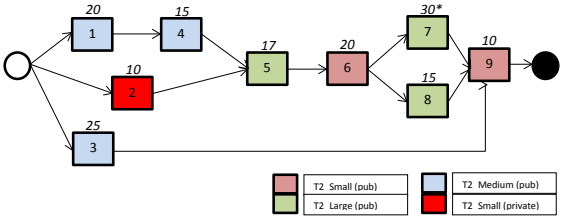

Figure 6: The second workflow input model.

on VMs. Thus, the execution cost would be the product of the number of partial hours consumed and the price of the VM. In this simulation, we use a subset of Amazon EC2 VM pricing. Table 1 shows the list of the VM types used and their prices as offered by Amazon (at the time of writing) in the US-East region. While Amazon prices are for public cloud VMs, we assume that a private version of those VMs with the same specifications would cost 10% more than their public counterpart. This is because private cloud requires in-house hardware and software maintenance, power, cooling, etc.

5.3 Simulation Results

We run the simulation 8 different times (with different configurations) and each run consists of 500 repetitions. The UFCFS algorithm is run once while the LFCFS is run three times (with the following values for the *limit*: 1,2 and 4). The pool-based adaptive task schedule algorithm is also run three times (with the following values for the *threshold*: 0.33, 0.5 and 0.75). And the proportional adaptive task schedule algorithm is run once.

During the simulation, the execution time of each individual workflow instance and the overall execution cost of all three instances were captured in each simulation run. In addition, we calculate the mean value of all the 500 repetitions. The simulation results are summarized in Table 2 where *W1*, *W2* and *W3* represent the execution times (in minutes) for workflows 1,2 and 3 respectively. In addition, *l* and *t* represent limit and threshold respectively. We can notice that (expectedly) the *UFCFS* algorithm gives the fastest execution but also the most expensive one. On the other hand, the *Proportional Adaptive Task Schedule* algorithm gives the best cost efficiency (23.3% cheaper than UFCFS), the best VM utilization and the

Table 1: Workflow engine VM types and prices.

EC2 Machine Type	Amazon (Public) Price ($)	Private Price ($)
T2_SMALL	0.026	0.0286
T2_MEDIUM	0.052	0.0572
T2_LARGE	0.104	0.1144
M4_LARGE	0.12	0.132

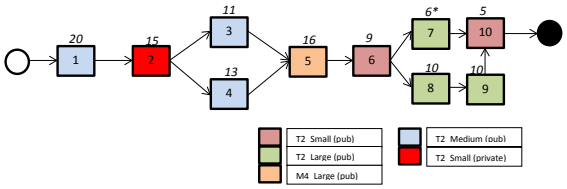

Figure 7: The third workflow input model.

second best overall execution time performance. Figure 8 shows a comparison between algorithms (and their parameter variation) in terms of execution cost.

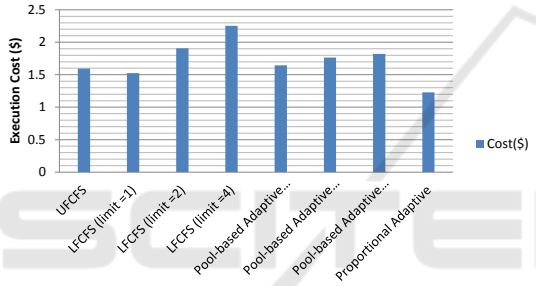

Figure 8: Execution cost benchmark in all algorithms.

Figure 9 shows the benchmark of all algorithms (the best performing parameter in the case of LFCFS and Pool-based Adaptive Task Schedule) for one of the input workflow models. The Proportional Adaptive Task Schedule gives the second best execution time (after the UFCFS). In the following subsections we detail the results further for each algorithm.

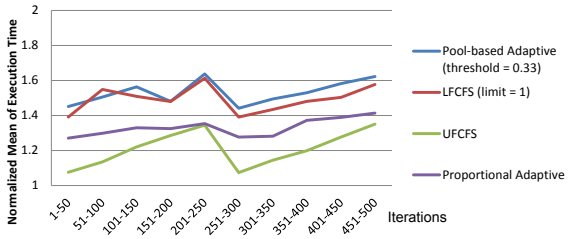

Figure 9: Execution time benchmark for all algorithms for workflow 3.

5.3.1 UFCFS

Figure 10 shows the normalized mean values for execution times of the three workflow input models. Normalization (which is applied to all of the following

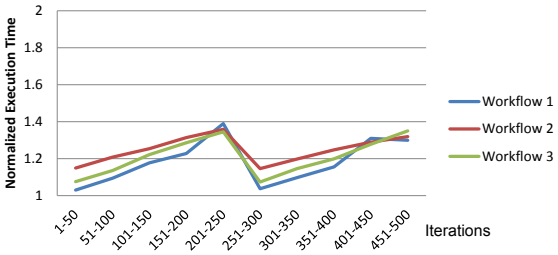

Figure 10: Execution times in UFCFS.

charts) is achieved by dividing each value by the minimum value in its category. In this chart (as well as other charts in this paper), the simulation runs have been grouped into groups of 50 runs and their mean was calculated. As the figure shows, UFCFS provides relatively low execution time since there are no delays required. However, the execution cost and the number of virtual machines used are relatively high as shown in Table 2.

5.3.2 LFCFS

We simulated LFCFS with three different *limit* values. From these simulations, we conclude that arbitrarily choosing the best value for the *limit* parameter is not possible as different values perform differently. The input models and their structure and complexity are among several factors that impact the results when using a particular *limit* value. Such factors are unpredictable, therefore, there is no systematic way for deciding the best arbitrary *limit* value to use. Figure 11 shows the normalized mean execution time for each workflow under the three different *limit* values as well as the normalized execution cost. We can clearly see that the cost increases linearly as the *limit* increases. In contrast, the execution times are reduced when the *limit* is higher.

5.3.3 Pool-based Adaptive Task Scheduling

By looking at the execution cost in Table 2 we see that the lower the *threshold*, the lower the execution cost as well. Finally, in Figure 12, we show the mean execution time for each workflow under different *threshold* values. We also show the overall execution cost. As expected, the higher the *threshold*, the faster and more expensive the execution. But again, there is

Table 2: Simulation results summary.

Algorithm	Parameters	W1	W2	W3	Cost ($)	VM No.
UFCFS	N/A	88.72	139.89	131.22	1.59	9.43
LFCFS	l = 1	200.43	253.26	208.46	1.52	5.88
LFCFS	l = 2	146.14	206.33	181.80	1.90	8.684
LFCFS	l = 4	125.58	194.65	170.89	2.25	11.00
Pool-based Adaptive	t = 0.33	193.93	254.03	208.13	1.64	5.83
Pool-based Adaptive	t = 0.5	176.09	233.25	201.59	1.76	6.56
Pool-based Adaptive	t = 0.75	165.18	217.07	193.51	1.81	7.28
Proportional Adaptive	N/A	144.06	184.15	147.19	1.22	5.81

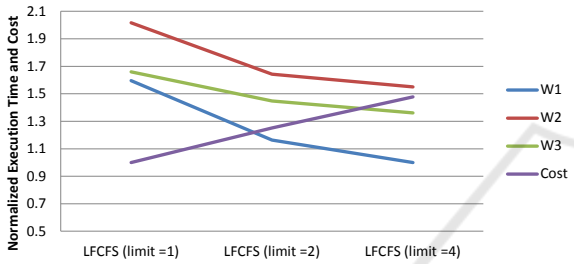

Figure 11: Execution time and cost benchmark in LFCFS.

no precise mechanism for finding the right trade-off point which also depends (in real situations) on unpredictable input workflow models.

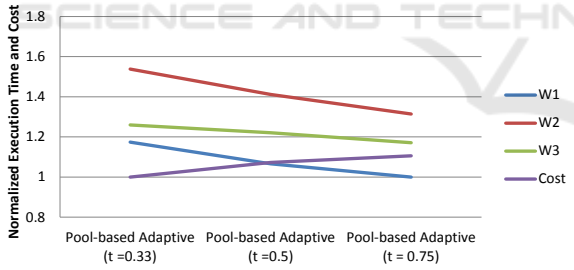

Figure 12: Execution times and cost in Pool-based Adaptive Task Schedule.

5.3.4 Proportional Adaptive Task Schedule

From Table 2 we can see that the Proportional Adaptive Task Schedule gives the best cost efficient schedule and the second best execution times. Figure 13 shows the three workflows execution times when scheduled using this algorithm. We can conclude that this algorithm is the most cost-efficient and provides the optimal workflows makespan among the four algorithms we presented. It is 23.28% cheaper than the UFCFS, 19.74% cheaper than the best LFCFS variation and 25.61% cheaper than the best Pool-based Adaptive Task Schedule variation. Additionally, we notice that the Proportional Adaptive Task Schedule algorithm is the most efficient from a resource utilization point of view (almost twice as efficient as the UFCFS).

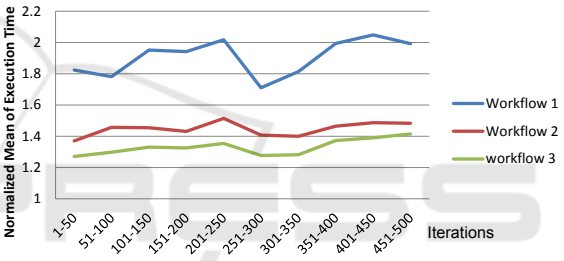

Figure 13: Execution times in Proportional Adaptive Task Schedule.

6 CONCLUSION

In this paper, we have highlighted the need for cost-aware scheduling of software processes workflows in the cloud. We have shown that software processes workflows contain different types of activities compared to scientific and business workflows and that the state-of-the-art scheduling algorithms do not meet the needs of executing software processes workflows. To meet these needs, we adapted three algorithms; the Unlimited First Come First Serve (UFCFS), Limited First Come First Serve (LFCFS) and the Pool-based Adaptive Task Schedule. We also proposed a fourth one; the Proportional Adaptive Task Schedule. We evaluated their performance (through simulation) in terms of overall execution cost and the makespan of individual workflow instances. The simulation results show that the UFCFS gives the shortest makespan while our proposed Proportional Adaptive Task Schedule gives the most cost-effective schedule, the best resource utilization and the second best

makespan. Unlike the LFCFS and the Pool-based Adaptive Task Schedule, the Proportional Adaptive Task Schedule does not rely on any arbitrary values and balances between the execution cost and the makespan.

The algorithms presented in this paper target the cloud-based software process scheduling problem in the context of the SDaaS architecture. However, they can be applied to similar problems which require resource-constrained project scheduling (RCPSP) (ZDAMAR and ULUSOY, 1995) and job-shop scheduling problem (JSSP) (Applegate and Cook, 1991).

In the future, we plan to perform larger experiments with larger process models (derived from real software processes). These experiments will target matching the PSBLIB (Kolisch and Sprecher, 1997) benchmark for the RCPSP problem which is categorized into 30, 60, 90, and 120 activity sets. Further, we plan to increase the number of iterations to 1000 plus to show the performance of the algorithms.

REFERENCES

Alajrami, S., Gallina, B., and Romanovsky, A. (2016a). EXE-SPEM: Towards Cloud-based Executable Software Process Models. In *MODELSWARD'16 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February.*, pages 517–526. Scitepress.

Alajrami, S., Romanovsky, A., and Gallina, B. (2016b). Software Development in the Post-PC Era: Towards Software Development as a Service. In Abrahamsson, P. and Jedlitschka, A., editors, *The 17th International Conference on Product-Focused Software Process Improvement, PROFES'16, Trondheim, Norway, November 22-24, Proceedings*. Springer.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156.

Bala, A. and Chana, I. (2011). Article: A Survey of Various Workflow Scheduling Algorithms in Cloud Environment. *IJCA Proceedings on 2nd National Conference on Information and Communication Technology*, NCICT(4):26–30.

Fuggetta, A. (2000). Software Process: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering Co-located with the International Conference on Software Engineering ICSE'00, Limerick, Ireland, June 4-11*, pages 25–34. ACM.

Fuggetta, A. and Di Nitto, E. (2014). Software Process. In *Proceedings of the on Future of Software Engineering, FOSE'14, Hyderabad, India, May 31 - June 7*, pages 1–12. ACM.

Jang, S., Wu, X., Taylor, V., Mehta, G., Vahi, K., and Deelman, E. (2004). Using performance prediction to allocate grid resources. Technical Report GriPhyN Project, TR 2004-25, Texas A&M University, USA.

Kolisch, R. and Sprecher, A. (1997). Psplib - a project scheduling problem library. *European Journal of Operational Research*, 96(1):205 – 216.

Liu, K., Jin, H., Chen, J., Liu, X., Yuan, D., and Yang, Y. (2010). A Compromised-Time-Cost Scheduling Algorithm in SwinDeW-C for Instance-Intensive Cost-Constrained Workflows on Cloud Computing Platform. *International Journal of High Performance Computing Applications*, 24:445–456.

Mao, M. and Humphrey, M. (2011). Auto-scaling to Minimise Cost and Meet Application Deadlines in Cloud Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 12-18, SC'11, Seattle, WA, USA*, pages 49:1–49:12. ACM.

OMG (2008). *Software and Systems Process Engineering Meta-Model Specification, V2.0*. Number formal/2008-04-01. Object Management Group (OMG), MA, USA.

Paulk, M., Curtis, W., Chrissis, M. B., and Weber, C. (1993). Capability maturity model for software (version 1.1). Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie Mellon University.

Singh, L. and Singh, S. (2013). Article: A Survey of Workflow Scheduling Algorithms and Research Issues. *International Journal of Computer Applications*, 74(15):21–28.

Vijindra and Shenai, S. (2012). Survey on Scheduling Issues in Cloud Computing. *International Conference on Modelling Optimization and Computing, Procedia Engineering*, 38:2881 – 2888.

Wang, J., Korambath, P., Altintas, I., Davis, J., and Crawl, D. (2014). Workflow as a Service in the Cloud: Architecture and Scheduling Algorithms. *Procedia Computer Science*, 29:546 – 556.

Yu, J. and Buyya, R. (2005). A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49.

Yu, J., Buyya, R., and Tham, C. K. (2005). Cost-based scheduling of scientific workflow applications on utility grids. In Stockinger, H., Buyya, R., and Perrott, R., editors, *First International Conference on e-Science and Grid Computing (e-Science'05), December 5-8, Melbourne, Australia*, pages 140–147.

ZDAMAR, L. and ULUSOY, G. (1995). A survey on the resource-constrained project scheduling problem. *IIE Transactions*, 27(5):574–586.