Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds

Xiaomin Zhu, *Member, IEEE*, Ji Wang, Hui Guo, *Member, IEEE*, Dakai Zhu, *Member, IEEE*, Laurence T. Yang, *Senior Member, IEEE*, and Ling Liu, *Fellow, IEEE*

Abstract—Clouds are becoming an important platform for scientific workflow applications. However, with many nodes being deployed in clouds, managing reliability of resources becomes a critical issue, especially for the real-time scientific workflow execution where deadlines should be satisfied. Therefore, fault tolerance in clouds is extremely essential. The PB (primary backup) based scheduling is a popular technique for fault tolerance and has effectively been used in the cluster and grid computing. However, applying this technique for real-time workflows in a virtualized cloud is much more complicated and has rarely been studied. In this paper, we address this problem. We first establish a real-time workflow fault-tolerant model that extends the traditional PB model by incorporating the cloud characteristics. Based on this model, we develop approaches for task allocation and message transmission to ensure faults can be tolerated during the workflow execution. Finally, we propose a dynamic fault-tolerant scheduling algorithm, FASTER, for real-time workflows in the virtualized cloud. FASTER has three key features: 1) it employs a backward shifting method to make full use of the idle resources and incorporates task overlapping and VM migration for high resource utilization, 2) it applies the vertical/horizontal scaling-up technique to quickly provision resources for a burst of workflows, and 3) it uses the vertical scaling-down scheme to avoid unnecessary and ineffective resource changes due to fluctuated workflow requests. We evaluate our FASTER algorithm with synthetic workflows and workflows collected from the real scientific and business applications and compare it with six baseline algorithms. The experimental results demonstrate that FASTER can effectively improve the resource utilization and schedulability even in the presence of node failures in virtualized clouds.

Index Terms—Virtualized clouds, fault-tolerant scheduling, primary-backup model, overlapping, VM migration

1 Introduction

Cloud computing has become an enabling paradigm for on-demand provisioning of computing resources to dynamic applications' workload [1]. Virtualization is commonly used in cloud, such as Amazon's elastic compute cloud (EC2), to render flexible and scalable system services, and thus creating a powerful computing environment that gives cloud users the illusion of infinite computing resources [2]. Running applications on virtual resources, notably virtual machines (VMs), has been an efficient solution for scalability, cost-efficiency, and high resource utilization [3].

There are an increasing number of scientific applications in the areas such as astronomy, bioinformatics, and physics [4]. The ever-growing data and complexity of those applications make them demand a high-performance computing environment. Cloud as the latest distributed computing paradigm can offer an efficient solution.

Many scientific applications are of the real-time nature where the correctness depends not only on the computational results, but also on the time instants at which these results become available [5]. In some cases, it is necessary to guarantee the timeliness of applications. For instance, the workflows of weather forecasting and medical simulations have strict deadlines which, once being violated, can make the result useless [6]. Therefore, it is critical for these kinds of deadline-constrained applications to obtain guaranteed computing services even in the presence of machine failures. It is reported in [7] that for a system consisting of 10 thousand super reliable servers (MTBF of 30 years), there will still be one failure per day. Moreover, each year, about 1-5 percent of disk drives die and servers crash at least twice for a 2-4 percent failure rate. Note that, it is even worse that large-scale cloud providers such as Google, also use a large number of cheap commodity computers that may result in much more frequent failures [8]. As a consequence, delivering fault-tolerant capability in clouds, especially for real-time scientific workflows is critical and has become a hot research topic.

- *X. Zhu and J. Wang are with the Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha, Hunan, P. R. China 410073. E-mail: {xmzhu, wangji}@nudt.edu.cn.*
- *H. Guo is with the School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia. E-mail: huig@cse.unsw.edu.au.*
- *D. Zhu is with the Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249. E-mail: dzhu@cs.utsa.edu.*
- *L.T. Yang is with the Department of Computer Science, St. Francis Xavier University, Antigonish, NS B2G 2W5, Canada. E-mail: ltyang@stfx.ca.*
- *L. Liu is with the College of Computing, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: lingliu@cc.gatech.edu.*

Manuscript received 16 July 2015; revised 4 Feb. 2016; accepted 12 Mar. 2016. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by M. M. Hayat.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2543731

For real-time scientific workflows on clouds, scheduling plays an essential role for satisfying applications' requirements while efficiently utilizing system resources. Scheduling basically maps tasks in a workflow to machines such that deadlines and response time requirements are satisfied, even in the presence of hardware and software failures. To date, a lot of fault-tolerant scheduling strategies have been investigated (e.g., [5], [8], [13], [14], [17]) in the distributed system domain. One of the popular strategies for fault tolerance is using the primary-backup (PB, in short) model, in which two copies of one task are allocated on two different processing units.

Distinct from other computing environments such as clusters and grids, clouds have the following unique features: 1) Clouds use VMs (a physical host can have multiple VMs) as basic computational instances and allow VMs to migrate among multiple hosts. Tasks in a workflow are allocated to VMs instead of directly to physical hosts; 2) The resources assigned to run workflows can be dynamically changed at run-time according to the demand of workflows, i.e., a cloud can be scaled up to satisfy the increased resource requests and scaled down to improve the system's resource utilization when the demand is reduced.

These two unique features improve the scheduling flexibility, but also lead to increased scheduling complexity and difficulty, especially for the fault-tolerant scheduling using PB model. The challenge is two-fold: 1) A host's failure may result in multiple computational instances' (i.e., VMs') failures, which causes execution failure of a task if two copies of the task are all allocated to the failed VMs. It is, however, not the case for traditional PB fault-tolerant scheduling algorithms where the two copies of a task are mapped to two different processing units; 2) The VM migration in fault-tolerant scheduling algorithms with the PB model must consider extra constraints such as task execution precedence and data transmission order, to ensure that faults can be tolerated effectively.

To the best of our knowledge, no work has been done on fault-tolerant scheduling for real-time workflows on virtualized clouds. In this paper, we address this problem with the following contributions:

- We establish a real-time workflow fault-tolerant model on virtualized clouds, which extends the traditional PB fault-tolerant model by incorporating the cloud characteristics;
- We provide analytical strategies for task allocation and message transmission to support fault tolerant execution;
- We innovatively apply the overlapping and VM migration mechanisms to the task scheduling so that both fault tolerance and high resource efficiency can be achieved;
- We propose a resource elastic provisioning mechanism that has three merits: 1) enabling full use of the idle resource through a backward shifting scheduling method, 2) allowing fast resource provisioning through the vertical and horizontal resource scaling, and 3) avoiding unnecessary frequent resource allocation changes caused by fluctuated workflow requests;
- Based on our fault model, scheduling strategies, and resource provisioning scheme, we design a novel dynamic <u>f</u>ault-toler<u>a</u>nt <u>s</u>cheduling algorithm for real-<u>t</u>ime sci<u>e</u>ntific wo<u>r</u>kflows - FASTER to support them running on virtualized clouds.

The remainder of this paper is organized as follows. The next section reviews related work in the literature. Section 3 formally models the dynamic real-time fault-tolerant scheduling problem for scientific workflows on clouds. Section 4 analyzes the constraints of task allocation and message transmission. Our FASTER algorithm and its supporting principles are discussed in Section 5. Section 6 presents the experiments and performance analysis using synthetic tasks and tasks from real-world traces. Section 7 concludes this paper and points out our future work.

2 Related Work

Since occurrences of faults are often unpredictable in computer systems, fault tolerance must be taken into consideration when designing scheduling algorithms [9]. There are two fundamental and widely recognized techniques that are able to support dynamic fault-tolerant scheduling in distributed environment: resubmission and replication [6]. So far as the resubmission is concerned, it resubmits a task to the system after a fault occurs in the resource on which the task was allocated. For example, Plankensteiner and Prodan suggested a resubmission heuristic to meet soft deadlines even in the absence of historical failure trace information and models [10]. Dean and Ghemawat employed the resubmission technique when designing the MapReduce in which if a handle worker—a computational unit—fails, it will be reset back to its initial state and the tasks allocated on it will be rescheduled to other workers [11]. Resubmission normally leads to much late finish time for tasks, and may cause them to miss their deadlines.

On the other hand, the replication approach makes multiple copies of a task and allocate each copy to a different resource to guarantee the successful completion of the task before its deadline, even in the presence of some resource failure. Basically, the more copies are allocated, the higher fault-tolerant capability of the system, which, nonetheless, may incur large resource consumption. Thereby, the two-copy replication (also known as the primary-backup model, or PB in short), has gained its popularity. With the PB model, a task gets only two copies: primary copy and backup copy [12].

In order to improve system schedulability while providing fault tolerance with low overhead, many studies have concentrated on overlapping techniques when using the PB model. Currently, there are two overlapping schemes: backup-backup overlapping (BB overlapping in short, in which multiple distinct backup copies are allowed to overlap with each other on the same computational unit) and primary-backup overlapping (PB overlapping in short, in which primary copies are allowed to overlap with other tasks' backup copies on the same computational unit). For example, Ghosh et al. employed a BB overlapping scheme allowing multiple backup copies overlap in the same time slot on a single processor; in their design, a deallocation scheme was used to release the resource reserved for backup copies after their corresponding primary copies finish successfully [13]. The work was extended for multiprocessor systems in [14], where processors are divided into groups to tolerate multiple simultaneous failures. Al-Omari et al. studied a PB overlapping policy for scheduling real-time tasks to achieve high schedulability [15]. Some research works combine both the BB and PB overlapping in scheduling algorithms. Sun et al. investigated a hybrid overlapping technique and provided a comprehensive redundance analysis on real-time multiprocessor systems [16]. Qin and Jiang considered the overlap constraints of the BB overlapping and PB overlapping for precedence constraint tasks and proposed an algorithm eFRD to enable a system's fault tolerance and maximize its reliability [5]. Zhu et al. also studied some fault-tolerant scheduling algorithms using hybrid overlapping schemes on clusters where QoS, reliability, adaptivity, timing constraint and resource utilization are considered [17], [18]. With the two-copy based approaches, the backup copy can be further implemented in two ways: passive backup and active backup. Passive backup copy starts to execute only when a fault occurs in its primary copy and the backup copy is deallocated once its primary copy finishes successfully (see examples in [14], [15]). Although this scheme is able to reduce the resource occupation, it cannot guarantee that all tasks meet their deadlines if some deadlines are relatively tight. In contrast, the active backup copy scheme allows the primary and backup copies of a task to execute simultaneously (see examples in [19], [20]). With the active backup copy, the probability of missing tasks' deadlines can be reduced, but the resource utilization will also be degraded. Therefore, some research made trade-off between the two schemes [17], [21]. Although the above methods consider real-time tasks (both dependent and independent), they do not take the emerging virtualization technology into account. They are suitable to some traditional distributed systems but not effective to the virtualized cloud computing environment.

Very recently, there appear some studies about scientific workflow scheduling in clouds. Mao and Humphrey studied workflow scheduling for systems with heterogeneous VMs (where each VM may have varied types and prices), to minimize the execution cost by applying a set of heuristics such as task merging [22]. Abrishami et al. proposed a static scheduling algorithm for a single workflow instance on a cloud. In their approach, all tasks on a partial critical path in the workflow were allocated to a single machine so as to minimize the execution cost [23]. Malawski et al. presented several static and dynamic scheduling algorithms to enhance the guarantee ratio of workflows while meeting QoS constraints such as budget and deadline. Also, they took the variant of tasks' execution time into account to enhance the robustness of their methods [24]. Rodriguez and Buyya suggested a resource provisioning and scheduling strategy for scientific workflows on IaaS cloud, in which the particle swarm optimization technique was employed to minimize the overall workflow execution within the timing constraint [25]. Calheiros et al. proposed a scheduling algorithm that uses the idle time of provisioned resources and budget surplus to replicate tasks, which efficiently mitigates the effects of performance variation of cloud resources on soft deadlines for workflows [26]. Unfortunately, all the work does not take the faults of machines into consideration while scheduling, thus they are not suitable for solving the fault tolerance issue on clouds. Zheng investigated the MapReduce fault tolerance in clouds and proposed heuristics to schedule backups, move backup instances, and select backups upon failure for fast recovery [8]. Plankensteiner and Prodan studied the fault-tolerant problem in clouds and proposed a heuristic that combines the task replication and task resubmission to increase the percentage of workflows that finish within soft deadlines [10]. The main distinction between their work and ours is three-fold. First, they do not consider the virtualization—the most unique feature of cloud—whereas our work takes VM as a basic computational unit. Second, elasticity is not considered in their work while our work allows the cloud scales dynamically based on the workload. Third, our work takes the overlapping technique into account to improve the resource utilization, which was not considered in previous work. In this paper, we present a novel dynamic fault-tolerant scheduling model and a scheduling algorithm FASTER for real-time scientific workflows executing on a virtualized cloud. Specifically, we apply the PB-based replication and the task overlapping, and exploit the cloud virtualization and elasticity in our approach to tolerating host failures.

3 System Model

In this section, we introduce the task and fault models and related notation and terminology used in this paper.

3.1 Task Model

A real-time scientific workflow with dependent tasks can be modelled by a Directed Acyclic Graph (DAG). In this paper, a DAG is defined as $G = \{T, E\}$, where $T = \{t_1, t_2, ..., t_n\}$ represents a set of real-time tasks that are assumed to be non-preemptive, and E is a set of directed edges that represents dependencies among tasks. An $e_{ij} = (t_i, t_j)$ in E indicates that task t_j depends on the data or message generated by task t_i for its execution, thus, t_j cannot start to execute until t_i finishes and the data or message yielded by t_i has been transferred to the location where t_j will execute. Task t_i is a parent of t_j and t_j is a child of t_i. For each task $t_i \in T$, we use $P(t_i)$ and $C(t_i)$ to denote its parents set and children set, respectively. $P(t_i) = \emptyset$ if t_i has no parents, and $C(t_i) = \emptyset$ if t_i has no children. Each workflow[1] G has an arrival time $a(G)$ and a deadline $d(G)$. A task $t_i \in T$, it can be modeled by $t_i = (a_i, d_i, s_i)$ where a_i, d_i, and s_i represent t_i's arrival time, deadline, and task size, respectively. Each task's deadline in a workflow can be calculated based on the workflow's deadline [5]. The task size is measured by Million of Instructions (MI), as used in [26], [27], [28]. With the PB model, each task t_i has two copies: primary copy t_i^P and backup copy t_i^B. They are allocated to two different hosts for fault tolerance. Given a task $t_i \in T$, s_i^P and f_i^P represent the start time and the finish time of t_i^P, respectively. Likewise, s_i^B and f_i^B denote the start time and the finish time of t_i^B. $P(t_i^P)$ and $P(t_i^B)$ represent the parents sets of t_i^P and t_i^B,

1. The terms workflow, job and DAG are used interchangeably throughout this paper.

respectively. And $C(t_i^P)$ and $C(t_i^B)$ represent the children sets of t_i^P and t_i^B, respectively.

We consider a virtualized cloud which contains a set $H = \{h_1, h_2, \cdots\}$ of unlimited number of physical computing hosts, to provide the hardware infrastructure for virtualized resources. The active host set is modeled by H_a with n elements, $H_a \subseteq H$. For a given host h_k, its processing capability p_k is characterized by its CPU performance in Million Instructions Per Second (MIPS). For each host $h_k \in H$, it contains a set $V_k = \{v_{1k}, v_{2k}, \cdots v_{|V_k|k}\}$ of virtual machines and each VM $v_{jk} \in V_k$ has the processing capability p_{jk} that is subject to $\sum_{j=1}^{|V_k|} p_{jk} \leq p_k$. The ready time of v_{jk} is denoted by r_{jk}.

In a virtualized cloud, one host may have one or multiple VMs on it and tasks are mapped on VMs instead of on hosts directly. In this study, we assume heterogeneous VMs; each VM can have different processing abilities. Therefore, the execution time of tasks' copies can be defined in the two execution time matrices E^P and E^B, where elements e_{ijk}^P and e_{ijk}^B specify the estimated execution time of task t_i^P on v_{jk} and task t_i^B on v_{jk}, respectively. We use x_{ijk}^P and x_{ijk}^B to reflect mappings of primary copies and backup copies to VMs at different hosts; x_{ijk}^P (x_{ijk}^B) is "1" if task t_i^P (t_i^B) is allocated to VM v_{jk}, otherwise, it is "0". Also, we use $v(t_i^P)$ and $v(t_i^B)$ to denote the VMs where t_i^P and t_i^B are allocated, $h(t_i^P)$ and $h(t_i^B)$ the corresponding hosts. Consequently, $x_{ijk}^P = 1$ means $v(t_i^P) = v_{jk}$, and $x_{ijk}^B = 1$ means $v(t_i^B) = v_{jk}$.

Considering the PB model, we use e_{ij}^{XY} to denote the edge between t_i^X and t_j^Y where $X, Y \in \{P, B\}$, i.e., t_i^X can be either t_i^P or t_i^B, and t_j^Y can be either t_j^P or t_j^B. For each edge e_{ij}^{XY}, there is an associated data transfer time tt_{ij}^{XY} that is the amount of time needed by t_j^Y from $v(t_i^X)$. If two tasks t_i^X and t_j^Y with dependence are assigned to the same host, the data transfer time $tt_{ij}^{XY} = 0$. Additionally, let dv_{ij} denote the transfer data volume between task t_i and t_j. Let $ts(h(t_i^X), h(t_j^Y))$ denote the transfer speed between $h(t_i^X)$ and $h(t_j^Y)$. Subsequently, we have $tt_{ij}^{XY} = \frac{dv_{ij}}{ts(h(t_i^X), h(t_j^Y))}$ when $h(t_i^X) \neq h(t_j^Y)$. The earliest start time est_j^Y of t_j^Y assigned to the VM v_{pq} can be calculated as:

$$est_j^P = \begin{cases} \max(a_j, r_{pq}) | x_{jpq}^P = 1 & \text{if } P(t_j^P) = \emptyset, \\ \max\limits_{t_i^X \in P(t_j^P)} (f_i^X + tt_{ij}^{XP}) & \text{otherwise.} \end{cases} \tag{1}$$

$$est_j^B = \begin{cases} \max(a_j, r_{pq}) | x_{jpq}^B = 1 & \text{if } P(t_j^B) = \emptyset, \\ \max\limits_{t_i^X \in P(t_j^B)} (f_i^X + tt_{ij}^{XB}, s_j^P) & \text{otherwise.} \end{cases} \tag{2}$$

The latest finish time lft_j^Y of t_j^Y is determined by the task's deadline, namely,

$$lft_j^Y = d_j. \tag{3}$$

The actual start time s_j^Y of task t_j^Y is the time at which the task is scheduled for execution. Task t_j^Y is able to be placed between est_j^Y and lft_j^Y if there exist slack time slots that can accommodate t_j^Y. To make the scheduling accurate and to achieve real-time guarantee, the completion time of a task will be sent to the scheduler after it is finished, and the VM resource information will be maintained and updated by the scheduler for facilitating the next round of scheduling decision making process.

One of the goals of our scheduling algorithm is to find suitable start time of tasks so that workflows can be processed as many as possible, hence achieving high overall throughput.

3.2 Fault Model

The fault tolerance problem addressed in this study is similar to those in [5], [8], [17] and is summarized below.

- Host failures are focused, which can trigger failures at other levels including VMs and applications.
- Failures on hosts are transient or permanent, and independent. Namely, a fault occurred on one host will not affect other hosts.
- Since the probability that two hosts fail simultaneously is small, we assume that at most one host fails at a time. The backup copies can be successfully finished if their corresponding primary copies are on the failed host.
- A fault-detection mechanism such as the fail-signal and acceptance test [13], [14] is available to detect host failures. New tasks will not be allocated to any known failed host.
- Reclamation mechanism [13] is obtainable. If the primary copy is finished successfully, the execution of the backup copy will be terminated and the resource reserved for the backup copy is reclaimed to improve resource utilization.

It should be noted that our fault model can be extended to tolerate multiple host failures by dividing hosts in a cloud into multiple groups, in each of which our fault-tolerant mechanism can be used like that described in [15].

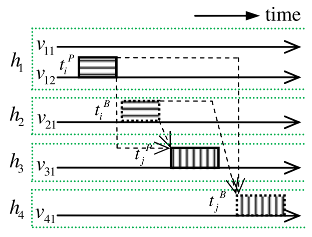
A key issue for the fault-tolerant scheduling using Primary/Backup model is to address the problem of how the allocation of tasks including primary copies and backup copies can guarantee the fault tolerance and what the impact of message transmission among dependent tasks may have on fault tolerance. Given that the constraints for fault tolerance guarantee are complex, in Section 4 we will provide a formal analysis on these two issues in detail.

4 Analysis of Task Allocation and Message Transmission

In this section, we analyze the scheduling conditions of tasks' allocation (including primary copies and backup copies) and message transmission when the PB model (for fault tolerance) and overlapping technique (for resource utilization efficiency) are applied. To facilitate the analysis, we firstly introduce some definitions.

Definition 1. Strong Primary Copy*: Given a task t_j^P, t_j^P is a strong primary copy if it is scheduled in such a way that when its host $h(t_j^P)$ is operational (without failure), t_j^P can always be executed.*

Take an example as shown in Fig. 1a where t_i is a parent of t_j, namely, t_j must receive message or data sent from t_i


(a) t_j^P is a strong primary copy

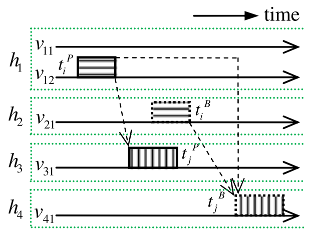

(b) t_j^P is a weak primary copy

Fig. 1. Examples of strong primary copy and weak primary copy. The dashed lines with arrows represent messages sent from parents to children.

for execution. As long as $h(t_j^P)$, i.e., h_3 is operational, t_j^P can be successfully executed because t_j^P is able to receive a message based on our assumption that at most one host fails at one time instant. Therefore, t_j^P is a strong primary copy.

Definition 2. Weak Primary Copy: *Given a task t_j^P, t_j^P is a weak primary copy if it is scheduled in such a way that it may not be executed even if $h(t_j^P)$ is operational.*

An example of weak primary copy is illustrated in Fig. 1b. Suppose $h(t_i^P)$, i.e., h_1 fails before f_i^P, then t_i^B has to execute. Since t_j^P cannot receive the message from t_i^B, t_j^P cannot execute even though $h(t_j^P)$, i.e., h_3 is operational. Therefore, t_j^P is a weak primary copy.

Based on the above definitions, we have the following proposition.

Proposition 1. *$\forall t_j \in T$, if t_j^P is a strong primary copy, it must belong one of the three cases: (1) $P(t_j) = \emptyset$; (2) $\forall t_i$, $t_i \in P(t_j)$, $h(t_i^P) \neq h(t_j^P)$, $s_j^P \geq f_i^P + tt_{ij}^{PP} \ \wedge \ s_j^P \geq f_i^B + tt_{ij}^{BP}$; (3) $\forall t_i$, $t_i \in P(t_j)$, $h(t_i^P) = h(t_j^P)$, $s_j^P \geq f_i^P + tt_{ij}^{PP}$; otherwise, t_j^P is a weak primary copy.*

The first case is straightforward from Definition 1. The second case has been demonstrated in Fig 1a. Here we show two examples for the third case in Fig. 2, where both primary copies are allocated to the same host and the backup copies to different hosts.

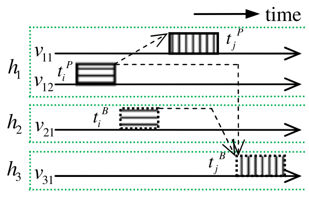
From Fig. 2, we can observe that whether t_j^P can receive a message from t_i^B or not, t_j^P is able to get messages from t_i^P and execute successfully since h_1 is assumed to be operational before f_j^P.

4.1 Basic Constraints for Dependent Tasks in a DAG

We now analyze the scheduling constraints between parent tasks and child tasks when fault tolerance is considered. We assume $t_i, t_j \in T$, $t_i \in P(t_j)$ and $t_j \in C(t_i)$.

Lemma 1. *For two dependent tasks t_i and t_j, t_i is the parent of t_j, if t_i^P finishes successfully, t_i^P must send the resulting message to t_j^P and t_j^B.*

Proof. We prove the lemma by contradiction. If t_i^P finishes successfully, according to the fault model (see Section 3.2), its backup t_i^B should be cancelled and the message path from the parent task t_i to t_j^B is broken. Suppose that t_i^P only sends message to t_j^P. If t_j^P fails, the backup t_j^B cannot execute, which makes the scheduling invalid. Contradiction happens. Therefore, a message from t_i^P must be sent to both t_j^P and t_j^B. □


(a) $s_j^P > f_i^B$

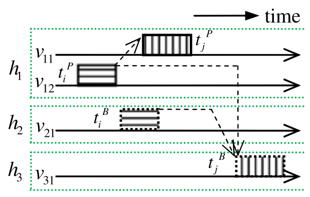

(b) $s_j^P < f_i^B$

Fig. 2. Examples of the third case in Proposition 1.

Lemma 1 specifies the transmission connections from a parent task. However, for a child task, the connections to its parents can be varied, depending on the parent type and their host allocations, which are discussed below.

4.1.1 When t_i^P is a Strong Primary Copy

If t_j is a child of t_i, t_j^P can be either a strong primary copy or a weak primary copy, and either $h(t_i^P) \neq h(t_j^P)$ or $h(t_i^P) = h(t_j^P)$. Therefore, there are four cases.

Case 1. t_j^P is a strong primary copy and $h(t_i^P) \neq h(t_j^P)$. Fig. 3a shows an example of this case.

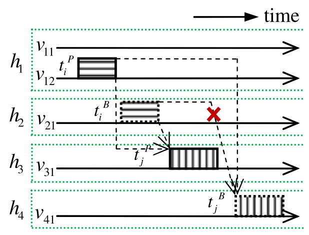
From Fig. 3a, we can observe that the edge e_{ij}^{BB} is redundant, namely t_j^B does not require the message from t_i^B. According to Lemma 1, the edges e_{ij}^{PP} and e_{ij}^{PB} are needed. If the edge e_{ij}^{BB} is required, t_i^B may need to execute. If t_i^B executes, $h(t_i^P)$ must fail before f_i^P, then other hosts should work (based on our fault model). Hence t_j^P should get executed. Consequently, the edge e_{ij}^{BB} is redundant.

By eliminating the edge e_{ij}^{BB}, the start time of t_j^B can be shifted to an earlier time, which increases the probability of finishing t_j^B before its deadline. The earliest start time of t_j^B can be recalculated as follows:

$$est_j^B = \max\Big\{s_j^P, f_i^P + tt_{ij}^{PB}\Big\}. \tag{4}$$

Case 2. t_j^P is a weak primary copy and $h(t_i^P) \neq h(t_j^P)$. Fig. 1b shows an example of this case.

In this case, edges e_{ij}^{PP}, e_{ij}^{PB}, and e_{ij}^{BB} are not redundant. Otherwise, fault tolerance cannot be achieved. It should be noted that t_j^B must have two edges connected with both t_i^P and t_i^B unlike t_j^P that has only one edge with t_i^P because it is


(a) t_j^P is a strong primary copy

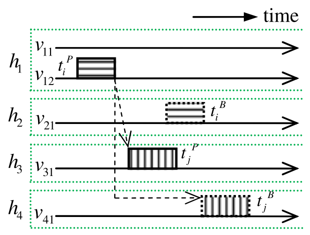
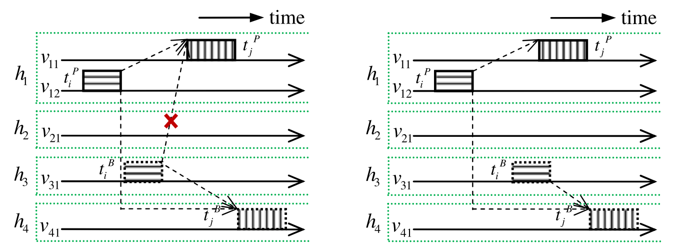

(b) t_j^P is a weak primary copy and t_j^B has no edge with t_i^B

Fig. 3. Examples of $h(t_i^P) \neq h(t_j^P)$.



(a) t_j^P is a strong primary copy (b) t_j^P is a weak primary copy

Fig. 4. Examples of $h(t_i^P) = h(t_j^P)$.

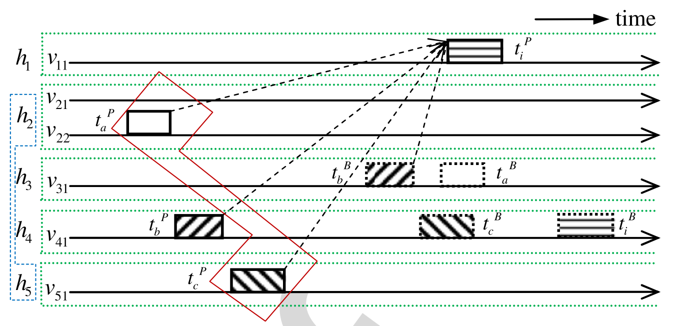


Fig. 5. An example that $\Delta_i\{\cdot\}$, $\Delta_i^P\{\cdot\}$, and $HS(\Delta_i^P\{\cdot\})$. The solid line that encircles primary copies represents the set $\Delta_i^P\{\cdot\}$; the dashed line that encircles hosts represents the set $HS(\Delta_i^P\{\cdot\})$.

a weak primary copy. Fig. 3b illustrates an example in which t_j^B only receives a message from t_i^P (Only two dependent tasks t_i and t_j are considered). From Fig. 3b, we can find that if $h(t_i^P)$ fails before f_i^P, neither t_j^P nor t_j^B can execute. Thus, t_j^B must connect to t_i^B, which gives the earliest start time of t_j^B as below:

$$est_j^B = \max\Big\{s_j^P, f_i^P + tt_{ij}^{PB}, f_i^B + tt_{ij}^{BB}\Big\}. \tag{5}$$

From the above analysis in Case 1 and Case 2, we can get the following propositions.

Proposition 2. *Given two tasks $t_i, t_j \in T$, $t_i \in P(t_j)$, $t_j \in C(t_i)$, if t_i^P is a strong primary copy and $h(t_i^P) \neq h(t_j^P)$: (1) if t_j^P is a strong primary copy, then t_j^P must have edges with t_i^P and t_i^B (i.e., e_{ij}^{PP} and e_{ij}^{BP}), and t_j^B must have an edge with t_i^P (i.e., e_{ij}^{PB}); (2) if t_j^P is a weak primary copy, then t_j^P must have an edge with t_i^P (i.e., e_{ij}^{PP}), and t_j^B must have edges with t_i^P and t_i^B (i.e., e_{ij}^{PB} and e_{ij}^{BB}).*

Proposition 3. *Given two tasks $t_i, t_j \in T$, $t_i \in P(t_j)$, $t_j \in C(t_i)$, if t_i^P is a strong primary copy, t_j^P is a weak primary copy, and $h(t_i^P) \neq h(t_j^P)$, then t_j^B cannot be allocated to $h(t_i^P)$.*

In Proposition 3, it can be easily found that if $h(t_i^P) = h(t_j^B)$, when the host fails, t_j^P also cannot execute due to the nature of weak primary copy. Therefore, t_j^B cannot be allocated to $h(t_i^P)$.

Case 3. t_j^P is a strong primary copy and $h(t_i^P) = h(t_j^P)$. Fig. 4a shows an example of this case.

In Fig. 4a, the edge e_{ij}^{BP} is redundant. Based on Lemma 1, the edges e_{ij}^{PP} and e_{ij}^{PB} are needed. If the edge e_{ij}^{BP} was required, both t_i^B and t_j^P would have to execute. t_i^B executes on condition that $h(t_i^P)$ fails before f_i^P, which means that t_j^P cannot execute. Therefore, when t_i^B executes, it only needs to send results to t_j^B and edge e_{ij}^{BP} is redundant.

Case 4. t_j^P is a weak primary copy and $h(t_i^P) = h(t_j^P)$. Fig. 4b depicts an example of this case.

By Lemma 1, the edges e_{ij}^{PP} and e_{ij}^{PB} are needed. Besides, the edge e_{ij}^{BB} is required when $h(t_i^P)$ fails. The earliest start time of t_j^B in Case 3 and Case 4 can be calculated as Eq. (5).

Based on above analysis, we can get the following Proposition 4.

Proposition 4. *Given two tasks $t_i, t_j \in T$, $t_i \in P(t_j)$, $t_j \in C(t_i)$, if t_i^P is a strong primary copy and $h(t_i^P) = h(t_j^P)$, t_j^P must have an edge with t_i^P (i.e., e_{ij}^{PP}), and t_j^B must have edges with t_i^P and t_i^B (i.e., e_{ij}^{PB} and e_{ij}^{BB}).*

4.1.2 When t_i^P is a Weak Primary Copy

When t_i^P is a weak primary copy, the constraint analysis becomes complicated. We have the following three definitions to facilitate the analysis.

Definition 3. Set of Tasks that Cause Weak Primary Copy $\Delta_i\{\cdot\}$: *a set of tasks that are parents of a task t_i and t_i^P cannot receive messages from those tasks' backup copies.*

Definition 4. Set of Primary Copies of Tasks that Cause Weak Primary Copy $\Delta_i^P\{\cdot\}$: *a set of primary copies of tasks that in the set $\Delta_{\{\cdot\}}$.*

Definition 5. Set of Hosts Accommodating Primary Copies of Tasks that Cause Weak Primary Copy $HS(\Delta_i^P\{\cdot\})$: *a set of hosts accommodating primary copies of tasks that in the set $\Delta_i\{\cdot\}$.*

Take an example as shown in Fig. 5. t_i has three parents t_a, t_b, and t_c. However, only t_a and t_c make t_i^P a weak primary copy. Thus, $\Delta_i\{\cdot\}$ becomes $\Delta_i\{a, c\} = \{t_a, t_c\}$. $\Delta_i^P\{a, c\} = \{t_a^P, t_c^P\}$. $HS(\Delta_i^P\{a, c\}) = \{h(t_a^P), h(t_c^P)\} = \{h_2, h_5\}$.

Lemma 2. *Given a weak primary copy t_i^P, its corresponding backup copy t_i^B cannot be allocated to the host in $HS(\Delta_i^P\{\cdot\})$.*

Proof. By contradiction. Assume t_i^B is allocated to a host h_k that is in $HS(\Delta_i^P\{\cdot\})$, $t_k^P \in \Delta_i^P\{\cdot\}$, and $h\{t_k^P\} = h_k$. If h_k fails before f_k^P, t_i^P cannot receive a message from t_k^P, so t_i^B has to execute. However, t_i^B has been allocated to h_k resulting in failures of both t_i^P and t_i^B - a contradiction. □

Now, we consider the constraints in terms of task allocation of t_i's children. Suppose $t_j \in C(t_i)$, t_j^P may be a strong primary copy or a weak primary copy. In addition, $h(t_i^P) \neq h(t_j^P)$ or $h(t_i^P) = h(t_j^P)$. Thereby, we analyze the constrains under the following four cases.

Case 1. t_j^P is a strong primary copy and $h(t_i^P) \neq h(t_j^P)$. Fig. 6 shows an example of this case.

Theorem 1. *Given two task $t_i, t_j \in T$, $t_j \in C(t_i)$, if t_i^P is a weak primary copy, t_j^P is a strong primary copy, and $h(t_i^P) \neq h(t_j^P)$, then (1) t_j^P cannot be allocated to any host in $HS(\Delta_i^P\{\cdot\})$, or*

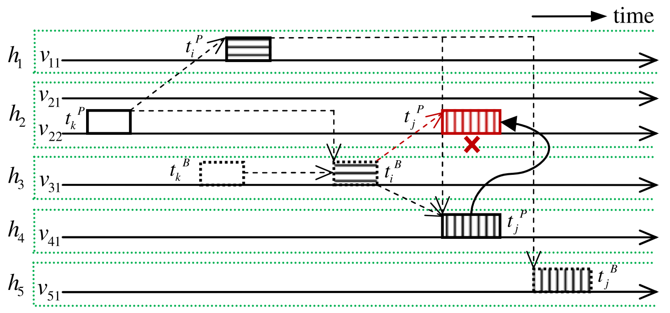


Fig. 6. An example that t_i^P is a weak primary copy, t_j^P is a strong primary copy, and $h(t_i^P) \neq h(t_j^P)$. t_j^P cannot be allocated to $h(t_k^P)$ if there is no edge e_{ij}^{BB}.

(2) when t_j^P is allocated to a host in $HS(\Delta_i^P\{\cdot\})$, an edge e_{ij}^{BB} should be added.

Proof. We prove this theorem by contradiction. Suppose t_j^P has been allocated to a host h_k in $HS(\Delta_i^P\{\cdot\})$, $t_k^P \in \Delta_i^P\{\cdot\}$, $h\{t_k^P\} = h_k$, and there is no edge e_{ij}^{BB}. If h_k fails before f_k^P, t_i^P cannot execute, so t_i^B executes successfully based on Lemma 2. Since t_j^P has been allocated to h_k, t_j^P cannot execute, thus t_i^B must send a message to t_j^B. Unfortunately, there is no edge e_{ij}^{BB}. A contradiction occurs. □

Case 2. t_j^P is a weak primary copy and $h(t_i^P) \neq h(t_j^P)$. Fig. 7 shows an example of this case.

Case 3. t_j^P is a strong primary copy and $h(t_i^P) = h(t_j^P)$.

Case 4. t_j^P is a weak primary copy and $h(t_i^P) = h(t_j^P)$.

Fig. 8 shows an example of case 3 and case 4.

Theorem 2. *Given two tasks $t_i, t_j \in T$, $t_j \in C(t_i)$, and $h\{t_k^P\} = h_k$. If t_i^P is a weak primary copy and t_j^P is a weak primary copy, then t_j^B cannot be allocated to any host in $HS(\Delta_i^P\{\cdot\})$.*

Proof. By contradiction. Suppose t_j^B has been allocated to a host h_k in $HS(\Delta_i^P\{\cdot\})$, $t_k^P \in \Delta_i^P\{\cdot\}$, and $h\{t_k^P\} = h_k$. If h_k fails before f_k^P, t_i^P cannot execute, and thus t_j^P cannot execute. Based on Lemma 2, t_i^B executes and t_j^B must execute after receiving a message from t_i^B. However, t_j^B has been allocated to h_k, t_j^B also cannot execute. A contradiction happens. Therefore, t_j^B cannot be allocated to h_k. □

Theorem 3. *Given two tasks $t_i, t_j \in T$, $t_j \in C(t_i)$, if t_i^P is a weak primary copy, t_j^P is a strong primary copy, and $h(t_i^P) = h(t_j^P)$,*

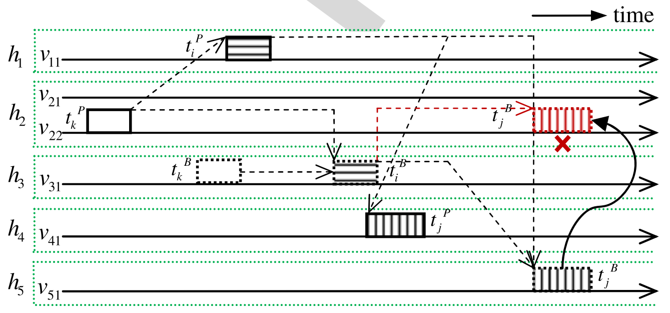


Fig. 7. An example that t_i^P is a weak primary copy, t_j^P is a weak primary copy, and $h(t_i^P) \neq h(t_j^P)$. t_j^B cannot be allocated to $h(t_k^P)$.

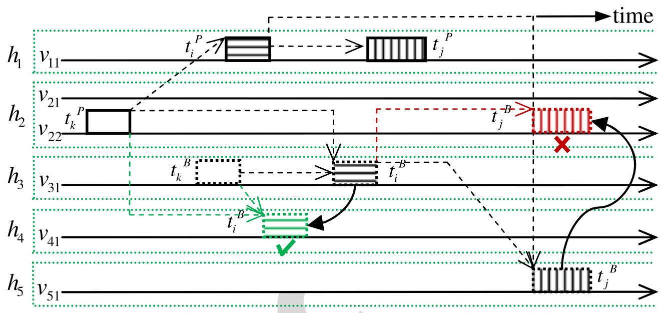


Fig. 8. An example that t_i^P is a weak primary copy and $h(t_i^P) = h(t_j^P)$. t_j^B cannot be allocated to $h(t_k^P)$ whether t_j^P is a strong primary copy or not.

then (1) t_j^B cannot be allocated to any host in $HS(\Delta_i^P\{\cdot\})$, or (2) when t_j^B is allocated to a host in $HS(\Delta_i^P\{\cdot\})$, an edge e_{ij}^{BP} should be added.

Proof. Suppose t_j^B is allocated to a host h_k in $HS(\Delta_i^P\{\cdot\})$, $t_k^P \in \Delta_i^P\{\cdot\}$, $h\{t_k^P\} = h_k$, and there is no edge e_{ij}^{BP}. If h_k fails before f_k^P, t_i^P fails to execute, thus t_i^B can execute based on Lemma 2. Because no edge e_{ij}^{BP} exists, t_j^B has to execute. Nevertheless, t_j^B has been allocated to h_k, t_j cannot be finished. A contradiction happens. Hence, t_j^B cannot be allocated to h_k. □

4.2 Overlapping Mechanisms

Schedulability can be improved by efficiently utilizing the cloud resources. Based on our fault model, a majority of backup copies do not execute. We therefore employ the overlapping mechanism to schedule tasks.

For the independent tasks in a DAG, the overlapping can be done based on our previous work [17]. In this paper, we focus on the overlapping mechanism and constraints for dependent tasks in a DAG.

Unlike the independent tasks, the backup-backup (BB) overlapping will be prohibited for dependent tasks (refer to [5], [8] for more explanation). However, the primary-backup (PB) overlapping is permitted. Fig. 9 shows two scenarios of PB overlapping.

Fig. 9 shows two possible overlapping examples. In both cases, t_i and t_j can be successfully executed even in the presence of a host's failure. In Fig 9a, when t_i^P finishes successfully, t_i^B will be cancelled, and t_j^P can execute. Regarding Fig. 9b, when t_i^P successfully finishes, t_i^B is cancelled and t_j^P can start to execute, without timing conflict. Also, it is easy to find that this kind of overlapping can achieve fault tolerance.

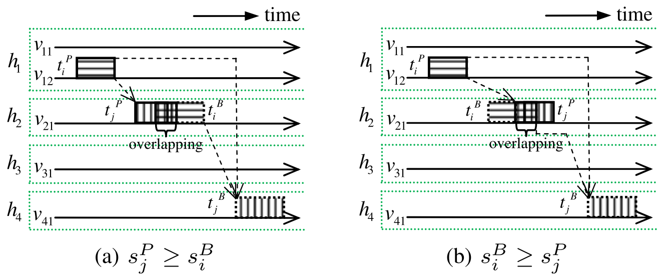


Fig. 9. Examples of PB overlapping.

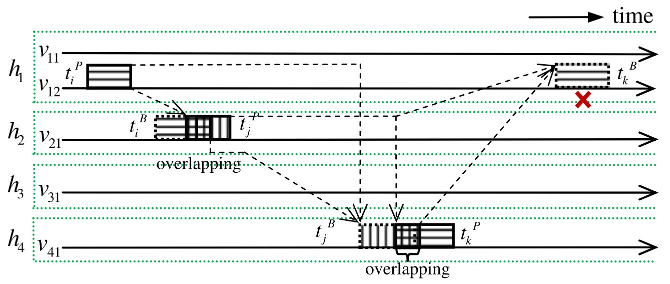

time
h_1 v_{11} v_{12} t_i^P t_k^B
h_2 v_{21} t_i^B t_j^P
overlapping
h_3 v_{31}
h_4 v_{41} t_j^B t_k^P
overlapping

Fig. 10. An example of Proposition 5. t_k^B cannot be allocated to h_1, h_2, and h_4.

Definition 6. Set $OHS\{\cdot\}$: *a set of hosts on which overlapped tasks (either primary copies or backup copies) are allocated.*

An overlapped task is the task that has its one copy overlapped with a copy of another task. For example in Fig. 10, task t_i overlaps with task t_j and task t_j also overlaps with task t_k. The three tasks are overlapped tasks. The related $OHS\{\cdot\}$ is $\{h_1, h_2, h_4\}$. We have a constraint of task allocation imposed by the PB overlapping.

Proposition 5. $\forall t_k \in T$, *if* t_k^P *overlaps with a backup copy whose host is in* $OHS\{\cdot\}$, *then* t_k^B *cannot allocated to any host in* $OHS\{\cdot\}$.

Fig. 10 illustrates an example of Proposition 5. In Fig. 10, t_k^P overlaps with t_j^B, t_k^B cannot be allocated to h_1, h_2 and h_4. For instance, if t_i^P fails before f_i^P, t_i^B executes, thus t_j^P cannot execute, and eventually leading to invoking t_k^B. However, t_k^B cannot execute due to the failure of h_1. Similar results will happen when h_2 fails.

4.3 Constraints of VM Migration

VM migration is an efficient approach to consolidating VMs so as to improve resource utilization and energy efficiency in clouds. Apart from the constraints (propositions, lemmas and theorems) discussed above, extra conditions should be satisfied in the VM migration in order to guarantee fault tolerance.

Proposition 6. *Assume* $NH\{\cdot\}$ *is a host set on which a task copy (primary copy or backup copy) cannot be allocated, the VM where the task is sitting on cannot be migrated to* $NH\{\cdot\}$.

Take an example as shown in Fig. 11 that is derived from Fig. 6. In Fig. 6, t_j^P cannot be allocated to h_2. Consequently, v_{41} cannot be migrated to h_2.

Since the message transmission time may be varied after VM migration, a task on another host perhaps miss its deadline, thus the VM migration must guarantee the timing constraint as below.

Proposition 7. *Suppose* $v(t_j^X)$ *is migrated.* $\forall t_i \in P(t_j), t_k \in P(t_i)$, $f_i^X + tt_{ij}^{XX'} + e_j^X > d_j$, $f_j^{X'} + tt_{jk}^{XX'} + e_k^X > d_k$. *where* $tt_{ij}^{XX'}$ *and* $tt_{jk}^{XX'}$ *are new transmission time and* $f_j^{X'}$ *is new finish time of* t_j^X.

If the above constraints are followed, VM migration can be used in our scheduling algorithm while satisfying the real-time fault-tolerant requirements. Compared with traditional distributed systems, this technique is expected to exhibit a considerable advantage in terms of improving resource utilization.

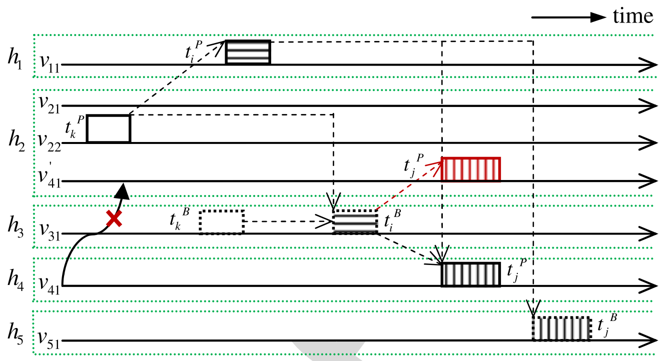


Fig. 11. An example of VM migration constraint. v_{41} cannot be migrated h_2 because t_j^P cannot be allocated h_2.

5 Fault-Tolerant Scheduling Algorithm – FASTER

Based on the aforementioned analysis, we develop a novel dynamic <u>f</u>ault-toler<u>a</u>nt <u>s</u>cheduling algorithm for real-<u>t</u>ime sci<u>e</u>ntific wo<u>r</u>kflows - FASTER to support virtualized clouds. FASTER consists of two parts: workflow scheduling and elastic resource provisioning. It provides high resource utilization while guaranteeing fault tolerance. When a workflow arrives, each task in the DAG will be forked into two copies, i.e., a primary copy and a backup copy. The workflows will be scheduled based on the First Come First Service policy, and the primary copy of a task will be scheduled before its backup copy. Different from the traditional work where as long as a task cannot be allocated successfully, the workflow will be rejected, our approach relaxes this strict restriction based on the fact that one single task's missing deadline does not mean the workflow will miss its deadline. If a parent task cannot be finish before its deadline, our FASTER strives to make its children tasks finished before their deadlines. If it is not possible, the workflow will then be rejected. Once a workflow is rejected, all the reserved resources by tasks in this DAG will be reclaimed.

Algorithm 1. Workflow Scheduling of FASTER

```
1  Estimate the deadline of each task based on the deadline of
   the workflow $G$;
2  $missDeadline \leftarrow \emptyset$;
3  while !all the task in G have been scheduled do
4      foreach $P(t_i) = \emptyset \,||\, P(t_i)$ have been scheduled do
5          success ← schedulingPrimary($t_i^P$);
6          success ← schedulingBackup($t_i^B$);
7          if !success then
8              if there exits a task $t_j$, $t_j \in P(t_i)$ &
               $t_j \in missDeadline$ then
9                  Reclaim the reserved resources;
10                 Reject the workflow $G$;
11             else
12                 $missDeadline \leftarrow missDeadline \cup \{t_i\}$;
13                 Re-calculate the possible earliest start time of
                   $C(t_i)$;
```

5.1 Workflow Scheduling

When a workflow arrives at the system, its tasks will be scheduled according to their precedence constraints. Algorithm 1 gives the pseudocode for fault-tolerant workflow scheduling.

Line 1 estimates the deadline of each task based on the deadline of the workflow. For a task that has no parents or its parents have been scheduled, Algorithm 1 attempts to first schedule its primary copy then its backup copy. The task is scheduled successfully only when both its primary copy and its backup copy are scheduled to be finished before its deadline. If one task misses its deadline, the system will bring its children's start time as early as possible (AEAP) (see Line 13) to prevent the child from missing its deadline. If two consecutive tasks miss their deadlines, the algorithm will reject the workflow and reclaim the reserved resources (see Lines 8-10).

Considering the fact that task scheduling in clouds is an NP-complete problem, we use a heuristic method to schedule primary copies and backup copies of tasks. In this work, we schedule tasks by inserting them into appropriate time slots based on our scheduling objectives. The following sections details the method of task insertion, primary copy scheduling and backup copy scheduling.

5.1.1 Task Backward Shifting

In order to make full use of the idle resources and execute new tasks as early as possible, it is desired that a new task is inserted into the idle time slot between two scheduled tasks. However, the idle time slot may be smaller than the execution time of new tasks, making use of the idle time not easy. Here we propose a task backward shifting method to improve the resource utilization, as explained below.

Definition 7. Backward Time Slack*: indicates how long the start time of a task can be shifted backward without any impact on the start time and the status (i.e., strong primary copy or weak primary copy) of the subsequent tasks.*

If a primary copy t_i^P will be shifted backward, the backward time slack bts_i^P of t_i^P is calculated as follows:

$$bts_i^P = \min\left\{ \min_{t_j \in C(t_i)} (s_j^P - tt_{ij}^{PP} - f_i^P), s_x - f_i^P \right\}, \tag{6}$$

where s_x denotes the start time of task t_x that is scheduled behind of t_i^P on the same VM, i.e., $v(t_i^P) = v(t_x)$. The first item in the right side of Eq. (6) guarantees the children of t_i can start on time and the second item ensures no delay of the following tasks scheduled on the same VM.

If a backup copy t_i^B will be shifted backward, the backward time slack can be calculated as follows:

$$\begin{aligned} bts_i^B &= \min\left\{ \min_{t_j \in C(t_i)} c_j, s_x - f_i^B \right\}, \\ c_j &= \begin{cases} s_j^P - tt_{ij}^{BP} - f_i^B, & \text{if } f_i^B + tt_{ij}^{BP} \le s_j^P \\ s_j^B - tt_{ij}^{BB} - f_i^B, & \text{if } f_i^B + tt_{ij}^{BP} > s_j^P \end{cases}, \end{aligned} \tag{7}$$

where the item $\min_{t_j \in C(t_i)} c_j$ in Eq. (7) ensure that the backward shifting will not affect the start time and status of the children of task t_i. The backward shifting of t_i^B may put the primary copy of its child, t_j, into two different types. The first case $f_i^B + tt_{ij}^{BP} \le s_j^P$ in Eq. (7) is when t_j^P is made as a strong primary copy. As a result, bts_i^B should not be larger than $s_j^P - tt_{ij}^{BP} - f_i^B$. The second case $f_i^B + tt_{ij}^{BP} > s_j^P$ is when t_j^P is made as the weak primary copy; In this case, it is not necessary to require t_i^B to finish before s_j^P. $s_j^B - tt_{ij}^{BB} - f_i^B$ insures that t_j^B can receive the result data of t_i^B before its start time.

By employing the backward shifting method, the system is able to reduce the actual system idle time and effectively increasing system resource utilization and performance.

5.1.2 Primary Copy Scheduling

In order to finish a task before its deadline, our algorithm strives to finish a primary copy as early as possible. Besides, to avoid that one host fault causes many primary copy failures, we attempt to distribute primary copies over all the active hosts so that each host has a similar number of primary task copies. The even distribution also increases the possibility of PB overlapping and hence enhances the resource utilization. The pseudocode for scheduling primary copies is detailed in Algorithm 2.

Algorithm 2. Function schedulingPrimary(t_i^P)

```
1  Sort $H_a$ in an increasing order by the count of scheduled primary copies;
2  $H_{candidate} \leftarrow$ top $\alpha$% hosts in $H_a$;
3  $eft \leftarrow +\infty; v \leftarrow NULL;$
4  while !all hosts in $H_a$ have been scanned do
5      foreach $h_k$ in $H_{candidate}$ do
6          if $h_k$ satisfies $t_i^P$'s scheduling constraints of Theorem 1 and Proposition 2 then
7              foreach $v_{kl}$ in $h_k.VmList$ do
8                  Calculate the earliest start time $est_i^P$ based on Eq. (1), Lemma 1, and Theorem 3;
9                  $eft_i^P \leftarrow est_i^P + e_{ikl}^P$;
10                 if $eft_i^P < eft$ then
11                     $eft \leftarrow eft_i^P$;
12                     $v \leftarrow v_{kl}$;
13     if $eft > d_i$ then
14         $H_{candidate} \leftarrow$ next top $\alpha$% hosts in $H_a$;
15     else
16         break;
17 if $eft > d_i$ then
18     if scale Up Resources($t_i^P$) then
19         return $true$;
20     else
21         Allocate $t_i^P$ to $v_{kl}$;
22         Update the $T_{shrink}$ and $T_{cancel}$ of $v_{kl}$;
23         return $false$;
24 else
25     Allocate $t_i^P$ to $v_{kl}$;
26     Update the $T_{shrink}$ and $T_{cancel}$ of $v_{kl}$;
27     return $true$;
```

Algorithm 2 first chooses the top α% hosts with fewer primary copies as the candidate hosts in Lines 1-2. Then, the VMs on these hosts are searched and the one that offers the earliest finish time for the primary copy is selected (see Lines 5-12). If no VM on the candidate hosts can finish the primary copy before its deadline, the next top α% hosts are chosen for the next round search (see Lines 13-16). By this method, a new primary copy is likely mapped to the hosts with fewer primary copies; hence overall primary copies can be evenly distributed among all the active hosts. If no existing VMs can accommodate the primary copy, the resource scaling-up mechanism will be performed in Line 18. If the scaling is not successful, Lines 21-23 attempt to schedule it on one VM and return the false value indicating that the primary copy cannot finish before its deadline. Note that although it misses deadline, our algorithm strives to bring forward its children's finish time to guarantee the DAG's deadline.

5.1.3 Backup Copy Scheduling

According to the analysis in Section 4, it can be found that a weak primary copy incurs more scheduling constraints than a strong primary copy. Proposition 1 also reveals that the main reason for a primary copy to become weak is that one backup copy of its parents cannot send results to the primary copy before its start time. As a result, differing to the widely used As Late As Possible strategy in the previous works [5], [17], our algorithm tries to finish a backup copy as early as possible to reduce the possibility of its children becoming weak primary copies. The algorithm also tries to aggregate backup copies to a smaller number of hosts so that when the system is reliable and few primary copies fail, most backup copies are cancelled, and the related hosts that mainly accommodate backup copies can be switched off. Algorithm 3 is the pseudocode for scheduling backup copies.

Algorithm 3. Function schedulingBackup(t_i^B)

```
1  $H_{candidate} \leftarrow$ the hosts on which no primaries are scheduled;
2  $H_{primary} \leftarrow$ Sort $H_a - H_{candidate}$ in an increasing order by the
   count of scheduled primaries;
3  $eft \leftarrow +\infty; v \leftarrow NULL;$
4  while !all hosts in $H_{primary}$ have been scanned do
5      foreach $h_k$ in $H_{candidate}$ do
6          if $h_k$ satisfies $t_i^B$'s scheduling constraints of Theorems 2,
           3, Lemma 2, and Propositions 3, 5 then
7              foreach $v_{kl}$ in $h_k.VmList$ do
8                  Calculate the earliest start time $est_i^B$ based
                   on Eqs. (2), (4), (5), Propositions 2, 4, and
                   Theorems 1, 3;
9                  $eft_i^B \leftarrow est_i^B + e_{ikl}^B$;
10                 if $eft_i^B < eft$ then
11                     $eft \leftarrow eft_i^P$;
12                     $v \leftarrow v_{kl}$;
13     if $eft > d_i$ then
14         $H_{candidate} \leftarrow$next top $\alpha\%$ hosts in $H_{primary}$;
15     else
16         break;
17 if $eft > d_i$ then
18     if scale Up Resources($t_i^B$) then
19         return $true$;
20     else
21         Allocate $t_i^B$ to $v_{kl}$;
22         Update the $T_{shrink}$ and $T_{cancel}$ of $v_{kl}$;
23         return $false$;
24 else
25     Allocate $t_i^B$ to $v_{kl}$;
26     Update the $T_{shrink}$ and $T_{cancel}$ of $v_{kl}$;
27     return $true$;
```

Following the steps similar to Algorithm 2, Algorithm 3 first chooses the candidate hosts with more backup copies in Lines 1-2. Then, the VMs on these hosts are scanned to find the one on which the finish time of backup copy is the earliest (see Lines 5-12). The resource scaling-up mechanism will function in line 18 to adjust the scale of resources when no existing VMs can accommodate the backup copy. If the resource scaling does not work, Lines 21-23 attempt to schedule it on one VM and return the false value indicating that the backup copy is not scheduled successfully.

5.2 Elastic Resource Provisioning

Elasticity is one of the most important characteristics of clouds. In this study, we incorporate it into our scheduling algorithm FASTER. The FASTER will adaptively add resources to accommodate tasks when the system is under a heavy workload, and decreases resources when the workload becomes light to enhance the resource utilization while guaranteeing fault tolerance.

5.2.1 Resource Scaling-Up

If a task's copy cannot be allocated to existing VMs, the resource scaling up mechanism will increase the processing capability of VMs or create a new VM to accommodate the new task. For a task t_i, the required processing capability p_r should satisfy the following formula:

$$est_i + s_i/p_r + delay < d_i, \tag{8}$$

where est_i is the earliest start time of task t_i which can be calculated by Eqs. (1) and (2), and $delay$ represents the time delay caused by the resource adjustment. If there is no VM satisfying the above requirement, a new VM, hence more resources, should be used. Scaling up can be done in two ways: the vertical scaling-up and the horizontal scaling-up.

The horizontal scaling-up creates a new VM with the required processing capability, and attempts to allocate it an existing active host. If the attempt fails, a sleep host will be turned on to accommodate the new VM. The horizontal scaling-up is easy to implement but suffers from a large delay from the VM creation and the host activation, which is unacceptable if tasks have tight deadlines. The vertical scaling-up approach temporarily increases the processing capability of an existing VM for the new task. In fact, with the virtualization technology, more and more real-world mainstream cloud platforms, such as OpenStack and CloudStack, support the live adjustment of VMs' processing capabilities, which is done with only a small, or even negligible performance overhead. The pseudocode for the resource scaling up mechanism is presented in Algorithm 4.

In Algorithm 4, the vertical resource scaling-up is first used in the mechanism. The active hosts are ordered by their remaining capacities. Lines 4-6 obtain the possible earliest start time for t_i on each of available VMs. Line 7 checks whether the remaining capacity of the host is sufficient for the VM to expand to the required processing capacity. If the vertical scaling up approach is feasible, Lines 8-9 map the task to the expanded VM, otherwise, the horizontal scaling up approach is called to create a new VM (see Lines 12-24). If it fails to create a suitable VM for the task to finish before its deadline, the algorithm will return false (see Line 24).

5.2.2 Resource Scaling-Down

To improve resource utilization, FASTER also incorporates a resource scaling-down scheme. Similar to scaling-up, the scheme consists of two scaling techniques: vertical scaling-down and horizontal scaling-down. The vertical scaling tries to shrink the processing capacity of an idle VM, and the horizontal scaling attempts to remove an idle VM. When a VM is idle for a relatively long time, the system first tries to shrink its processing capacity, and if the VM keeps idle for a certain time, it will be removed to improve resource utilization.

Algorithm 4. Function scaleUpResources(t_i^X)

```
1  Sort $H_a$ in a decreasing order by the remaining MIPS;
2  foreach $h_k$ in $H_a$ do
3      if $h_k$ satisfies scheduling constraints for $t_i^X$ then
4          foreach $v_{kl}$ in $h_k$ do
5              Calculate the earliest start time $est_i^X$;
6              Calculate the required processing capacity $p_r$ based
               on Eq. (8);
7              if $p_r - v_{kl}.MIPS \leq h_k.MIPS$ then
8                  Allocate $t_i^X$ to $v_{kl}$;
9                  Update the $T_{shrink}$ and $T_{cancel}$ of $v_{kl}$;
10                 Expand $v_{kl}.MIPS$ to $p_r$ during the execution time
                   of $t_i^X$;
11                 Return $true$;
12 Select a $newVm$ whose processing capacity satisfies Eq. (8);
13 foreach $h_k$ in $H_a$ do
14     if $h_k.MIPS \geq newVM.MIPS$ & $h_k$ satisfies scheduling con-
       straints for $t_i^X$ then
15         Create $newVM$ on $h_k$;
16         Allocate $t_i^X$ to $newVM$;
17         Return $true$;
18 Turn on a host $h_{new}$ in $H - H_a$;
19 if $h_{new}.MIPS \geq newVM.MIPS$ then
20     Create $newVm$ on $h_{new}$;
21     Allocate $t_i^X$ to $newVM$;
22     return $true$;
23 else
24     return $false$;
```

By introducing the vertical scaling-down approach, the processing capacity of idle VMs can be shrunk to the lowest level to reduce the resource waste, while when the system workload becomes heavy, the processing capacity of the shrunk VMs can expand in a relatively short time to accommodate the new tasks. Consequently, the system can adapt to the workload in a more flexible manner, and avoid unnecessarily creating or canceling VMs, especially when the submitted request fluctuates very frequently.

For each VM, we use T_{shrink} and T_{cancel} to respectively denote when the VM is expected to be shrunk and cancelled. We set T_{idle} and $T_{idle}^{'}$ for the VM shrinking time (to the VM's lowest processing capability) and VM canceling time. T_{shrink} and T_{cancel} can be calculated as follows,

- When a primary t_i^P is scheduled on the VM, $T_{shrink} = \max\{f_i^P + T_{idle}, T_{shrink}\}$, $T_{cancel} = \max\{f_i^P + T_{idle}^{'}, T_{cancel}\}$;
- When a backup t_i^B is scheduled on the VM, $T_{shrink} = \max\{f_i^P + T_{idle}, T_{shrink}\}$, $T_{cancel} = \max\{f_i^P + T_{idle}^{'}, T_{cancel}\}$; if t_i^B is due to execute because of host's fault, $T_{shrink} = \max\{f_i^B + T_{idle}, \ T_{shrink}\}$, $T_{cancel} = \max\{f_i^B + T_{idle}^{'}, T_{cancel}\}$.

Based on the above calculation method, the VM will be shrunk or cancelled if there is no task running on it for a period of T_{idle} or $T_{idle}^{'}$. Furthermore, because backup copies may be reclaimed if primary copies finish successfully, backup copies can be scheduled to finish or even start later than the time instant T_{shrink} and T_{cancel}, by which the system can effectively utilize the idle resources. Algorithm 5 shows the pseudocode for the resource scaling-down mechanism. And it runs independently in the scheduler by a thread without any calling from other algorithms.

Algorithm 5. Function scaleDownResources()

```
1  while true do
2      foreach VM $v_{kl}$ in the cloud system do
3          if it reaches the time $v_{kl}.T_{shrink}$ then
4              Shrink the processing capacity of $v_{kl}$ to $p_{lowest}$;
5          if it reaches the time $v_{kl}.T_{cancel}$ then
6              Remove $v_{kl}$ from $h_k$ and cancel it;
7              if $h_k.utilization \leq U_{low}$ then
8                  $offTag \leftarrow true$;
9                  foreach $v_{kl}$ in $h_k$ do
10                     $migTag \leftarrow false$;
11                     foreach $h_i$ in $H_a$ except $h_k$ do
12                         if $h_i$ can accommodate $v_{kl}$ & the migration
                           satisfies the constraints of Propositions 3, 5,
                           6, 7, Lemma 2, and Theorems 1, 2, 3 then
13                             $migTag \leftarrow true$;
14                             break;
15                     if $migTag == false$ then
16                         $offTag \leftarrow false$;
17                         break;
18                 if $offTag$ then
19                     Migrate VMs in $h_k$ to destination hosts;
20                     Switch $h_k$ to sleep status and remove it
                       from $H_a$;
21                 else
22                     Give up the migration operation;
```

When a VM reaches the time instant T_{shrink}, the processing capacity of this VM will be shrunk to the lowest level P_{lowest} to reduce the waste of resources (see Line 3). Further, if a VM reaches the time instant T_{cancel}, the VM will be cancelled. After that, if the host's capacity utilization falls below the lower threshold U_{low}, the system tries to consolidate the VMs on it to other hosts (see Lines 8-16), and then switches off the host to further enhance resource utilization (see Line 19).

6 Performance Study

To demonstrate the performance improvements gained by FASTER, a series of experiments on both a simulated cloud platform and a real virtualized cluster are conducted. We compare FASTER with five baseline algorithms: Non-Overlapping-FASTER (NOFASTER), Non-VM-Consolidation-FASTER (NCFASTER), Non-Vertical-Scaling-Up-FASTER (NVUFASTER), Non-Vertical-Scaling-Down-FASTER (NVDFASTER), and Non-Backward-Shifting-FASTER (NSFASTER). We also compare them with a classical fault-tolerant scheduling algorithm for dependent tasks, eFRD [5]. The main differences of these algorithms to FASTER and their uses are briefly described below.

- NOFASTER: no task overlapping. By comparing with NOFASTER, the effectiveness of the overlapping technique can be tested;
- NCFASTER: no resource consolidation. It can be used to test the improvement of resource utilization by using the scaling-down mechanism;
- NVUFASTER: no vertical scaling up. By comparing with NVUFASTER, the strength of vertical scaling-up can be evaluated;
- NVDFASTER: no vertical scaling down. Selecting NVDFASTER as a baseline algorithm is to demonstrate the performance improvements achieved by vertical scaling-down technique;
- NSFASTER: no backward shifting. It can be used to evaluate the effectiveness of backward shifting policy;
- eFRD: A classical fault-tolerant scheduling algorithm for dependent tasks [5]. It uses as early as possible strategy for both the primary copy scheduling and backup copy scheduling. However, it has no ability to dynamically adjust resources. To make the comparison fair, we slightly modify eFRD in such a way that it does not consider the reliability cost.

The performance metrics by which we evaluate the system performance are as follows:

- Guarantee Ratio (*GR*): the percentage of workflows (DAGs) that are guaranteed to finish successfully among all submitted workflows;
- Host Active Time (*HAT*): the total active time of all hosts in cloud, reflecting the resource consumption of the system;
- Ratio of Task time over Hosts time (*RTH*): the ratio of the total tasks' execution time over the total active time of hosts, reflecting the resource utilization of the system.

6.1 Experiments using Random Synthetic Workflows

Simulation for repeatability is used as the part of our experiments. In the simulations, the CloudSim toolkit [26]—a widely used cloud environment simulator in both industry and academia—is chosen as a simulation platform. We add some new settings to conduct our experiments. The detailed setting and parameters are given as follows:

Hosts are modeled with the processing capacity of 1,000, 1,500, 2,000 or 3,000 MIPS, and connected to 1 Gbps Ethernet. Four types of VMs with the processing power equivalent to 250, 500, 700 and 1,000 MIPS are considered. The time required for turning on a host and creating a VM is set as 90 and 15 s, respectively. Workflow arrival times follow the Poisson distribution with the average interval time $1/\lambda$ being uniformly distributed in the range $[\frac{1}{\lambda}, \frac{1}{\lambda+2}]$. The deadline of a workflow (DAG) is given as $d_i = a_i + \alpha \times e_i^{min}$, where e_i^{min} is the possible minimal execution time of this DAG, and α subjects to the uniform distribution, $U(1.5, 2.5)$. Each DAG is assumed to have random precedence constraints that are generated by the following steps similar to [5]:

- Choosing the number of tasks, N, for a workflow and the number of messages, M, that are passed between the N tasks. In our experiments, we set $N = 200$, and $M = \theta \times N$;
- Generating a size for each task in the DAG; the random task size is uniformly distributed in the range $[1 \times 10^5, 2 \times 10^5]$ MI;
- Randomly selecting a sender and a receiver for each message based on the condition that such selection does not generate communication loop in the workflow. The transfer data volume of this message is uniformly distributed in the range [10,100] MB;
- A deadline for each task is calculated according to the DAG deadline.

Table 1 gives the parameters and their values.

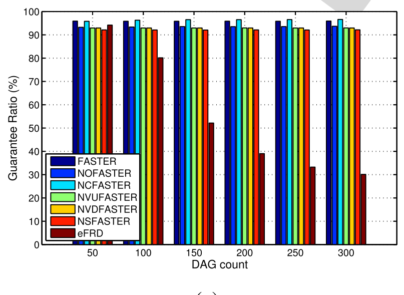
TABLE 1
Parameters of Workflows

Parameter	Value(Fixed)−(Min, Max, Step)
DAG count	(50)−(50, 300, 50)
Task size($\times 10^5$ MI)	([1, 2])
Interval time $1/\lambda$	(2)−(0, 10, 2)
α	(2)−(1.5, 2.5, 0.5)
θ	(4)−(2, 7, 1)

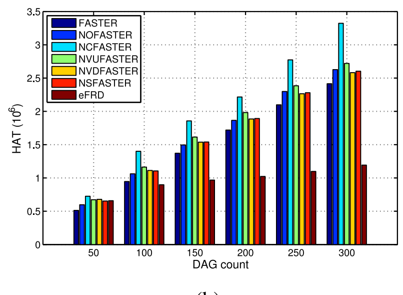
6.1.1 Performance Impact of DAG Count

In this section, we present a group of experiments to compare performance of the seven algorithms with varied number of workflows (i.e., DAG count).

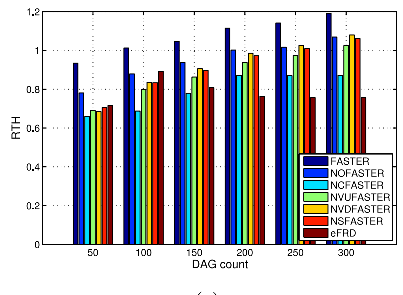
It can be observed from Fig. 12a that all the algorithms except eFRD basically maintain stable *GR*s for different DAG counts, which can be attributed to the fact that these six algorithms consider the infinite resources in the cloud, thus when DAG count increases, new hosts will be dynamically available for more DAGs. While eFRD assumes fixed available hosts and has no ability to adjust resource, so with



(a)



(b)



(c)

Fig. 12. Performance impact of DAG count.

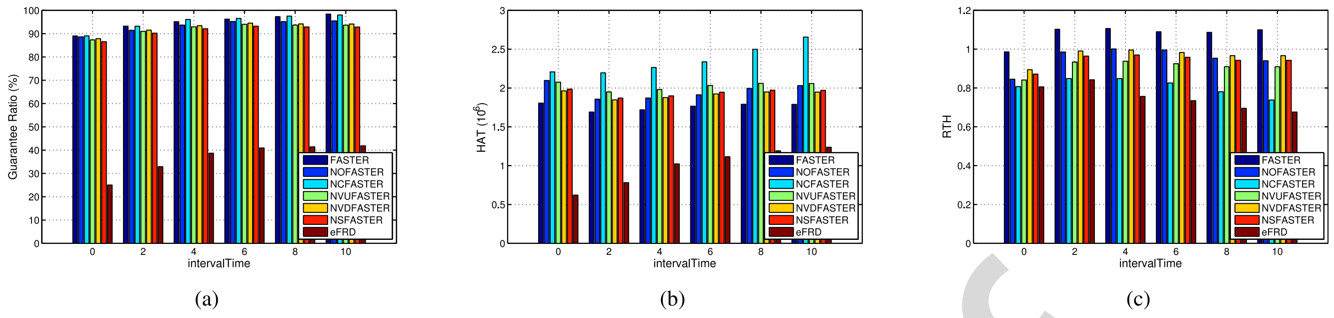


Fig. 13. Performance impact of arrival rate.

the increase of DAG count, the GR of eFRD decreases correspondingly. Since no overlapping technique is employed by NOFASTER, more resources are occupied by backup copies. Consequently, the GR of NOFASTER is lower than that of FASTER. Also, NSFASTER has a lower GR than FASTER, it is because NSFASTER does not use the backward time slack method, resulting in allocation failures for some tasks in a DAG. Although NCFASTER has the high GR similar to FASTER, it uses more resources due to lack of resource consolidation, as can be seen from Fig 12b—NCFASTER has the highest Host Active Time (HAT).

From Fig. 12b, it can be seen that FASTER maintains the lowest HAT among all algorithms except eFRD, which indicates those integrated policies in FASTER can work well to improve the resource utilization. In addition, because there is no consolidation mechanism employed in NCFASTER, some resources are in idle state, resulting in the highest resource consumption; this consumption also increases rapidly with the DAG count. The second big resource consumer is VNUFASTER. This is because NVUFASTER does not effectively reuse existing active VMs (namely, vertical scaling-up) for new tasks; it can only accommodate more new tasks by adding new VMs (horizontal scaling), which inevitably increases the hosts active time. In contrast, our vertical scaling-up mechanism can effectively reduce the resource usage. The high HAT from NSFASTER also demonstrates the effectiveness of the backward shifting techniques used in FASTER.

It can be seen from Fig. 12c that the resource utilization increases with the DAG count for all algorithms except for eFRD. For eFRD, its RTH increases first and then decreases, which can be explained below. With eFRD, the system resource is assumed fixed. The resource is sufficient enough to accommodate most DAGs for small number of workflows (in the range from 50 to 100). The task running time increases with the DAG count, hence the resource utilization increases. But the system becomes saturated when DAG count is further increased (DAG count $>$ 100), making the host running longer and therefore lowing the resource utilization. We can also observe from Fig. 12c that among other algorithms NCFASTER has the lowest RTH due to lack of resource consolidation. FASTER, on the other hand, incorporates the mechanism to consolidate resources and hence achieves the highest RTH.

6.1.2 Performance Impact of Arrival Rate

In this section, we inspect the impact of the workflow arrival rate on the scheduling performance. The related experimental results are given in Fig. 13.

From Fig. 13a, we can see that eFRD has a lower GR than other FASTER related algorithms since those algorithms can dynamically scale up system with extra resources for more tasks. It can also be seen that the GRs from those algorithms increase with the $intervalTime$. It is because the smaller $intervalTime$ means the heavier system workload, leading to more VMs being created. Since the creation of VMs causes the delay which may result in some workflows missing their deadlines. With the increase of the $intervalTime$, the need to create new VMs is reduced, hence more tasks can be finished within the deadline. In addition, similar to Figs. 12a, Fig. 13a shows that FASTER and NCFASTER have higher GRs than other algorithms—for the same reason as given for Fig. 12a.

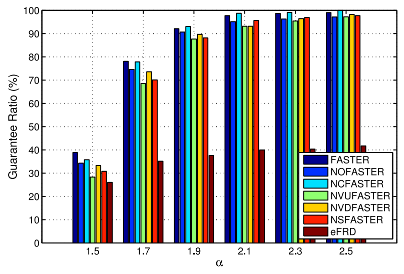
From Fig. 13b, we can see that among all FASTER related algorithms, FASTER always keeps the resource consumption (HAT) lower than other algorithms when the $intervalTime$ varies. The worst performer is NCFASTER and its HAT becomes more and more significant than others when the $intervalTime$ increases. Additionally, when $intervalTime$ is 0, namely, a burst of DAGs surge into the system, FASTER can effectively retain the resource overhead (through overlapping of backup copies), achieving a much lower HAT than NOFASTER.

Similarly, from Fig. 13c we can observe the highest performance of FASTER in terms of resource utilization (RTH) under different task arrival rates. In contrast, the performances of NCFASTER and eFRD are quite poor, which is understandable since NCFASTER does not consider resource consolidation and eFRD cannot dynamically adjust the resource usage; when the task arrival $intervalTime$ increases, more resources become idle, hence the cloud resource utilization is decreased.

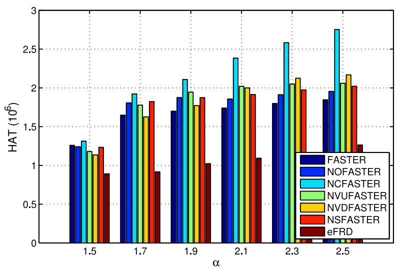
6.1.3 Performance Impact of Deadline

To investigate the impact of deadline on the performance, we ran the experiments with the deadline parameter α varying from 1.5 to 2.5. High α values indicate the tighter deadlines. The related results are shown in Fig. 14.

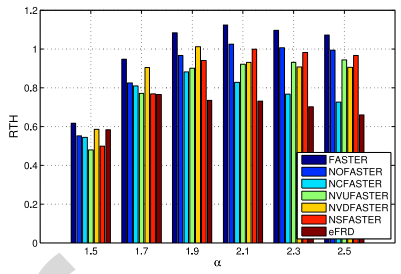
As can be seen from Fig. 14a, when the deadline is very tight (i.e., $\alpha = 1.5$), there is not enough time for resources to scale up and most DAGs cannot be processed, hence GRs are quite low for all FASTER related algorithms, particularly for NVUFASTER where the vertical scaling-up is not implemented. The vertical scaling-up takes less time than horizontal scaling and thus can response to the system workload quickly. Lacking the vertical scaling-up mechanism results in low schedulability, especially in a tight



(a)



(b)



(c)

Fig. 14. Performance impact of deadline.

deadline case. With the increase of α (namely, the deadline becomes loose), the GRs of all algorithms are improved.

Fig. 14b shows that with the increase of α, the HATs of all algorithms increase. This is because more DAGs are accepted, requiring more host active time to finish these DAGs. Notably, the increase speed of HAT by NCFASTER is obviously faster than others, which indicates that NCFASTER cannot sufficiently utilize active host resources without the consolidation mechanism and needs more hosts to finish DAGs compared with other algorithms. Moreover, the HAT of NVDFASTER is larger than those of others except NCFASTER. This is because NCFASTER does not use VM vertical scaling and the cloud system cannot be scaled down immediately, resulting in more idle resources.

The advantage of FASTER is again shown in Fig. 14c. FASTER has the highest resource utilization. When deadline is tight (e.g., $\alpha = 1.5$), NCFASTER and NVDFASTER also have higher RTHs. This can be explained that the resource scale-down will hardly occur when deadline is very tight, thus lacking VM consolidation mechanism and VM vertical scaling-down mechanism will not greatly affect the cloud resource utilization. However, when deadline becomes looser, the RTHs of NCFASTER and NVDFASTER are obviously inferior to others. Regarding eFRD, its RTH is better when deadline is tight because almost all the resources are used to enhance the resource utilization whereas when deadline becomes looser, some resources are idle resulting in decreased RTH.

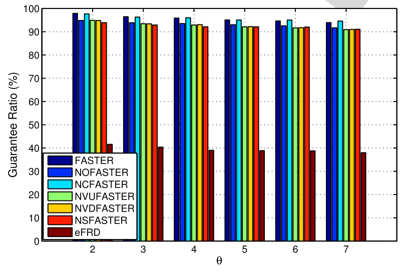
6.1.4 Performance Impact of Task Dependence

In this section, we examine the impact of task dependent degree in DAGs on the performance. To measure the dependence between tasks in a workflow, we use the number of messages that are passed between tasks. We set the message number $M = \theta \times N$. The bigger the θ is, the more message connections between tasks, hence the higher the task dependence. Parameter θ varies from 2 to 7. Fig. 15 shows the experimental results.

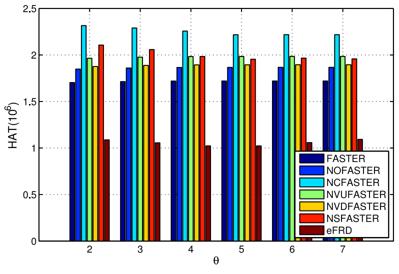
From Fig. 15a, we can observe that with the increase of θ, the GRs of all the algorithms slightly decrease. This is because the increased task dependency degrades the system schedulability. It can also be seen that similar to other experiments (Figs 12a and 13a) FASTER and NCFASTER offer the highest GRs.

Fig. 15b shows that NCFASTER and NSFASTER have higher HATs than other algorithms, which is especially evident for small θ (i.e., low task dependence). With the small task dependence, more tasks in a DAG can be executed in parallel and can be finished earlier; hence some hosts can be freed and shut down earlier. However, such an advantage is neither exploited by NCFASTER (no resource consolidation mechanism) nor effectively used by NSFASTER (no backward shifting scheme). Hence they consume more resources. When θ increases, the tasks that can be executed in parallel decrease, thus the difference of HATs between them (NCFASTER and NSFASTER) and other algorithms becomes smaller.

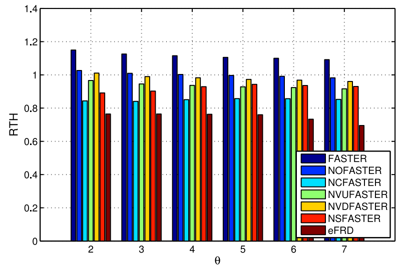
It can be observed from Fig. 15c that FASTER maintains the highest RTH. With the increase of θ, the RTHs of all the algorithms except NSFASTER decrease. This is because high task dependence increases execution constraints, making less tasks available to VMs even if some of them are idle, hence reducing resource utilization. When the value of θ becomes larger, the tasks that can be executed in parallel decrease, hence less resource will be wasted leading to lower RTH.



(a)



(b)



(c)

Fig. 15. Performance impact of task dependence.

TABLE 2
Experimental Results Using Trace-Based Workflows

Metr. Alg.	FASTER	NOFASTER	NCFASTER	NVUFASTER	NVDFASTER	NSFASTER	eFRD
GR (%)	98.5%	96.0%	99.0%	96.5%	97.5%	95.0%	29.0%
HAT ($\times 10^6 s$)	3.02	3.34	4.26	3.59	3.61	3.56	1.45
RTH	0.95	0.85	0.65	0.77	0.76	0.75	0.58

6.2 Experiments Using Trace-Based Workflows

In order to evaluate the feasibility of our proposed algorithm to the real applications, we further test it based on the DAG models using five real applications: LIGO, Montage, CyberShake, Epegenomics and SIPHT. For each application, we use the Workflow Generator [29] to create jobs with the size of 50, 100, 200 and 500 tasks. For each job size, 20 different DAG instances are generated based on the real workflow traces [4]. Hence, the total collection of synthetic DAGs contains five kinds of applications, four job sizes for each application, and 20 DAG instances, making a total of 400 synthetic DAGs.

In this trace-based experiment, 200 DAGs are submitted to the cloud system at the rate following Poisson distribution with the average interval time 4. The calculation method of the DAG's deadline is similar to that for random synthetic workflows discussed in the above section. To reflect the diversity of DAGs in clouds, the DAG is selected randomly among the 400-DAG set generated.

From the results listed in Table 2, it can be found that FASTER performs better than the others. Compared with the guarantee ratios of the random synthetic workflows in the previous section, those in the trace-based experiment are much better, especially FASTER and NCFASTER by which nearly all the DAGs can be scheduled successfully. This is because the precedence constraints of real applications are much lower than those of random synthetic workflows. There are a large number of independent tasks in the real applications that can be processed in parallel with new VMs. For eFRD, its guarantee ratio is lower than that of random synthetic workflows due to the lack of the resource adjustment mechanism in eFRD. The large number of parallel tasks cannot be finished timely by limited computing resources. This result from the real application again demonstrates that the resource adjustment mechanism is essential for schedulability.

It can also be seen that the *HAT*s in this group of experiment are larger than those in the previous experiments. This is due to the fact that the average execution time of tasks in real applications is larger than that in random synthetic workflows.

From Table 2, we can see that FASTER has high resource utilization in the trace-based experiment. It outperforms NCFASTER and NSFASTER by up to 46.15 and 26.67 percent. FASTER achieves much higher *RTH* from the trace-based experiment than from the synthetic-workflow based experiment, which also can be attributed to the large numbers of parallel tasks in real applications. These parallel tasks cause many new VMs to be created in the system. When they are finished, the VMs become idle. However, for NCFASTER, the idle hosts cannot be switched off immediately, wasting of resources. For NSFASTER, the differences of the finish times between parallel tasks are more evident when the count of parallel tasks increases. Without the backward shifting technology, the VMs that finish tasks earlier will be idle and wait for other tasks, and as a result the computing resources are wasted. From the above results in the trace-based experiments, it can be concluded that FASTER can effectively improve schedulability and resource utilization for real applications.

7 Conclusions and Future Work

This paper investigates the fault-tolerant scheduling problem for real-time scientific workflows in virtualized clouds. The scheduling goal is to improve the system's schedulability and cloud resource utilization while tolerating hardware failures. The fault-tolerant capability of our FASTER algorithm is realized through an extended primary-backup model that integrates the virtualization and elasticity—characteristics of clouds. We thoroughly studied the constraints of task allocation and message transmission for PB based workflow model in a virtualized cloud. For high system resource utilization, the elastic resource provisioning, one of the most important features of clouds, is considered and instrumented with overlapping and VM migration techniques. In addition, a backward shifting method is used to allow FASTER to make full use of the idle resources. More importantly, the vertical and horizontal scaling-up strategies are both employed in FASTER for fast resource provisioning when workflows burst into a cloud in a very short period. Furthermore, FASTER adopts a vertical scaling-down approach to avoiding the ineffective resource scaling when the submitted request fluctuates very frequently.

To evaluate the performance of FASTER, we conduct extensive experiments with random synthetic workloads and real workflows to compare it with six baseline algorithms: NOFASTER, NCFASTER, NVUFASTER, NVDFASTER, NSFASTER, and eFRD. The experimental results show that FASTER provides good performance in both types of workloads. Based on the real-world trace used in our experiments, FASTER outperforms eFRD by 239.66 percent in terms of guarantee ratio, and by 63.79 percent in terms of resource utilization.

For future study, we will extend our fault-tolerant scheduling model to tolerate multiple hosts' failure. Communication faults will also be considered. To improve the scheduling accuracy, we will develop a prediction model with feedback for DAG execution time.

Acknowledgments

This research was supported by the National Natural Science Foundation of China under grants No. 61572511, No. 61403402, and No. 11428101, the Hunan Provincial Natural Science Foundation of China under grant No. 2015JJ3023 as well as the Southwest electron & telecom technology institute under Grant No. 2015014.

REFERENCES

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 57, no. 3, pp. 599–616, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A.Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] J. Rao, Y. Wei, J. Gong, and C. Xu, "QoS guarantees and service differentiation for dynamic cloud applications," *IEEE Trans. Netw. Serv. Manage.*, vol. 10, no. 1, pp. 43–55, Mar. 2013.

[4] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Gener. Comput. Syst.*, vol. 29, no. 3, pp. 682–692, 2013.

[5] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Comput.*, vol. 32, no. 5, pp. 331–356, 2006.

[6] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertesz, and P. Kacsuk, "Fault-tolerant behavior in state-of-the-art grid workflow management systems," Inst. On Grid Inf., CoreGRID-Netw. Excellence, Tech. Rep. TR-0091, 2007. Q1

[7] J. Dean, "Designs, lessons and advice from building large distributed systems," in *Proc. LADIS*, 2009. Q2

[8] Q. Zheng, "Improving MapReduce fault tolerance in the cloud," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, 2010, pp. 1–6.

[9] P. M. Alvarez and D. Mosse, "A responsiveness approach for scheduling fault recovery in real-time system," in *Proc. 5th IEEE Real-Time Technol. Appl. Symp.*, 1999, pp. 1–10.

[10] K. Plankensteiner and R. Prodan, "Meeting soft deadlines in scientific workflows using resubmission impact," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 5, pp. 890–901, May 2012.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] A. Amin, R. A. Ammar, and S. S. Gokhale, "An efficient method to schedule tandem of real-time tasks in cluster computing with possible processor failures," in *Proc. 8th IEEE Int. Symp. Comput. Commun.*, Jun. 2003, pp. 1207–1212.

[13] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 3, pp. 272–284, Mar. 1997.

[14] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 11, pp. 1137–1152, Nov. 1998.

[15] R. Al-Omari, A. K. Somani, and G. Manimaran, "Efficient overloading technique for primary-backup scheduling in real-time systems," *J. Parallel Distrib. Comput.*, vol. 64, no. 5, pp. 629–648, 2004.

[16] W. Sun, Y. Zhang, C. Yu, X. Defago, and Y. Inoguchi, "Hybrid overloading and stochastic analysis for redundant real-time multiprocessor systems," in *Proc. 26th IEEE Int. Symp. Rel. Distrib. Syst.*, 2007, pp. 265–274.

[17] X. Zhu, X. Qin, and M. Qiu, "QoS-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters," *IEEE Trans. Comput.*, vol. 60, no. 6, pp. 800–812, Jun. 2011.

[18] X. Zhu, C. He, R. Ge, and P. Lu, "Boosting adaptivity of fault-tolerant scheduling for real-time tasks with service requirements on clusters," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1708–1716, 2011.

[19] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A new fault-tolerant scheduling technique for real-time multiprocessor systems," in *Proc. 2nd Int. Worshop Real-Time Comput. Syst. Appl.*, 1995, pp. 197–202.

[20] C. H. Yang, G. Deconinec, and W. H. Gui, "Fault-tolerant scheduling for real-time embedded control systems," *J. Comput. Sci. Technol.*, vol. 19, no. 2, pp. 191–202, 2004.

[21] R. Al-Omari, A. K. Somani, and G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 595–608, 2005.

[22] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–12.

[23] S. Abrishami, M. Naghibzadeh, and D. Epema, "Deadline-constrained workflow scheduling algorithms for IaaS clouds," *Future Gener. Comput. Syst.*, vol. 23, no. 8, pp. 1400–1414, 2012.

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

[25] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 222–235, Apr.-Jun. 2014.

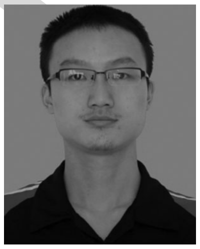
[26] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw.: Practice Exp.*, vol. 41, no. 1, pp. 23–50, 2011.

[27] M.-Y. Tsai, P.-F. Chiang, Y.-J. Chang, and W.-J. Wang, "Heuristic scheduling strategies for linear-dependent and independent jobs on heterogeneous grids," in *Proc. Int. Conf. Grid Distrib. Comput.*, 2011, pp. 496–505.

[28] S. Sadhasivam, N. Nagaveni, R. Jayarani, and R. V. Ram, "Design and implementation of an efficient two-level scheduler for cloud computing environment," in *Proc. Int. Conf. Adv. Recent Technol. Commun. Comput.*, 2009, pp. 884–886.

[29] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling and evaluating research on scientific workflows," in *Proc. E-Science*, 2014, pp. 177–184.

Xiaomin Zhu received the PhD degree in computer science from Fudan University, Shanghai, China, in 2009. He is currently an associate professor in the College of Information Systems and Management at the National University of Defense Technology, Changsha, China. His research interests include scheduling and resource management in distributed systems. He has published more than 70 research articles in refereed journals and conference proceedings such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Parallel and Distributed Computing*, and so on. He is a member of the IEEE.



Ji Wang received the BS degree in information systems from the National University of Defense Technology, China, in 2008. Currently, he is working toward the MS degree in the College of Information System and Management at National University of Defense Technology. His research interests include real-time systems, fault-tolerance, and cloud computing.

Hui Guo received the PhD degree in electrical and computer engineering from the University of Queensland. Prior to starting her academic career, she had worked in a number of companies/organizations for information systems design and enhancement. She is now a lecturer in the School of Computer Science and Engineering at the University of New South Wales, Australia. Her research interests include application specific processor design, low power system design, and embedded system optimization. She is a member of the IEEE.