Dynamic and Fault-Tolerant Clustering for Scientific Workflows

Weiwei Chen, *Student Member, IEEE,* Rafael Ferreira da Silva, Ewa Deelman, *Member, IEEE,* and Thomas Fahringer, *Member, IEEE*

Abstract—Task clustering has proven to be an effective method to reduce execution overhead and to improve the computational granularity of scientific workflow tasks executing on distributed resources. However, a job composed of multiple tasks may have a higher risk of suffering from failures than a single task job. In this paper, we conduct a theoretical analysis of the impact of transient failures on the runtime performance of scientific workflow executions. We propose a general task failure modeling framework that uses a Maximum Likelihood estimation-based parameter estimation process to model workflow performance. We further propose 3 fault-tolerant clustering strategies to improve the runtime performance of workflow executions in faulty execution environments. Experimental results show that failures can have significant impact on executions where task clustering policies are not fault-tolerant, and that our solutions yield makespan improvements in such scenarios. In addition, we propose a dynamic task clustering strategy to optimize the workflow's makespan by dynamically adjusting the clustering granularity when failures arise. A trace-based simulation of five real workflows shows that our dynamic method is able to adapt to unexpected behaviors, and yields better makespans when compared to static methods.

Index Terms—scientific workflows, fault tolerance, parameter estimation, failure, machine learning, task clustering, job grouping

1 Introduction

Scientific workflows can be composed of many fine computational granularity tasks, where the task runtime may be shorter than the system overhead—the period of time during which miscellaneous work other than the user's computation is performed. Task clustering methods [1]–[6] merge several short tasks into a single job such that the job runtime is increased and the overall system overhead is decreased. Task clustering is the most common technique used to address execution overheads and increase the computational granularity of workflow tasks executed on distributed resources [1]–[3]. However, existing clustering strategies ignore or underestimate the impact of failures on the system, despite their significant effect on large-scale distributed systems [7]–[10], such as Grids [11]–[14] and Clouds [11], [15], [16]. In this work, we focus particularly on transient failures since they are expected to be more prevalent than permanent failures [7].

A clustered job consists of multiple tasks. If a task within a clustered job fails (i.e., is terminated by unexpected events during its computation), the job is marked as failed, even though tasks within the same job have successfully completed their execution. Several techniques have been developed to cope with the negative impact of job failures on the execution of scientific workflows. The most common technique is to retry the failed job [17]–[19]. However, retrying a clustered job can be expensive since completed tasks within the job usually need to be recomputed, thereby resource cycles are wasted. In addition, there is no guarantee that recomputed tasks will succeed. As an alternative, jobs can be replicated to avoid failures specific to a worker node [20]. However, job replication may also waste resources, in particular for long-running jobs. To reduce resource waste, job executions can be periodically checkpointed to limit the amount of retried work. However, the overhead of performing checkpointing can limit its benefits [7].

In this work, we propose three fault-tolerant task clustering methods to improve existing task clustering techniques in a faulty distributed execution environment. The first method retries failed tasks within a job by extracting them into a new job. The second method dynamically adjusts the *granularity* or *clustering size* (number of tasks in a job) according to the estimated inter-arrival time of task failures. The third method partitions the clustered jobs into finer jobs (i.e., reduces the job *granularity*) and retries them.

We then evaluate these methods using a task transient failure model based on a parameter learning process that estimates the distribution of the task runtimes, the system overheads, and the inter-arrival time of failures. The process uses the Maximum Likelihood Estimation (MLE) based on *prior* and *posterior* knowledge to build the estimates. The prior knowledge about the parameters is modeled as a distribution with known parameters. The posterior knowledge about the task execution is also modeled as a distribution with a known *shape* parameter and an unknown *scale* parameter. The shape parameter affects the shape of a

- *W. Chen, R. Ferreira da Silva and E. Deelman are with the Information Sciences Institute, University of Southern California, Marina del Rey, CA, USA. T. Fahringer is with the Institute for Computer Science, University of Innsbruck, Innsbruck, Austria.*
- *Corresponding author: Weiwei Chen weiweich@google.com*

distribution, while the scale parameter affects the stretching or shrinking of a distribution.

The two first fault-tolerant task clustering methods were introduced and evaluated in [2] using two real scientific workflows. In this work, we extend our previous work by: 1) studying the performance gain of using fault-tolerant task clustering methods over an existing task clustering technique on a larger set of workflows (five widely used scientific applications); 2) evaluating the performance impact of the variance of the distribution of the task runtimes, the system overheads, and the inter-arrival time of failures on the workflow's makespan (turnaround time to execute all workflow tasks); and 3) characterizing the performance impact on the workflow's execution of dynamic and static failure estimations with different inter-arrival times of failures.

The rest of this manuscript is organized as follows. Section 2 gives an overview of the related work. Section 3 presents our workflow and task failure models. Section 4 details our fault-tolerant clustering methods. Section 5 reports experiments and results, and the manuscript closes with a discussion and conclusions.

2 Related Work

Failure analysis and modeling of computer systems have been widely studied over the past two decades. These studies include, for instance, the classification of common system failure characteristics and distributions [8], root cause analysis of failures [9], empirical and statistical analysis of network system errors and failures [10], and the development and analysis of techniques to prevent and mitigate service failures [21].

In scientific workflow management systems (WMS), fault tolerance issues have also been addressed. For instance, the Pegasus WMS [22] has incorporated a task-level monitoring system, which retries a job if a task failure is detected. Provenance data is also tracked and used to analyze the cause of failures [23]. A survey of fault detection, prevention, and recovery techniques in current grid WMS is available in [24]. The survey provides a compilation of recovery techniques such as task replication, checkpointing, resubmission, and migration. In this work, we combine some of these techniques with task clustering methods to improve the performance and reliability of *fine-grained* tasks. To the be best of our knowledge, none of the existing WMS have provided such features.

The low performance of fine-grained tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times at resources are high. Several papers have addressed the control of task granularity of loosely coupled tasks. For instance, Muthuvelu et al. [25] proposed a clustering algorithm that groups bag of tasks based on the runtime, and later based on task file size, CPU time, and resource constraints [26]. Recently, they proposed an online scheduling algorithm [27] that merges tasks based on resource network utilization, user's budget, and application deadline. In addition, Ng et al. [28] and Ang et al. [29] also considered network bandwidth to improve the performance of the task scheduling algorithm. Longer tasks are assigned to resources with better network bandwidth. Liu and Liao [30] proposed an adaptive scheduling algorithm to group fine-grained tasks according to the processing capacity and the network bandwidth of the currently available resources.

Several papers have addressed the workflow-mapping problem by using DAG scheduling heuristics [13], [31]–[34]. In particular, HTCondor [34] uses matchmaking to avoid scheduling tasks to compute nodes without sufficient resources (CPU power, etc.). Previously, we adopted a similar approach to avoid scheduling workflow tasks to compute nodes with high failure rates [2]. In this work, we focus on the performance gain of task clustering, in particular on how to adjust the clustering size to balance the cost of task retry and of the scheduling overheads.

The task granularity control has also been addressed in scientific workflows. For instance, Singh et al. [1] proposed a level- and label-based clustering. In level-based clustering, tasks at the same workflow level are clustered together. The number of clusters or tasks per cluster are specified by the user. In the label-based clustering, the user labels tasks that should be clustered together. Recently, Ferreira da Silva et al. [3], [5] proposed task grouping and ungrouping algorithms to control workflow task granularity in a non-clairvoyant and online context, where none or few characteristics about the application or resources are known in advance. Although these techniques significantly reduce the impact of the scheduling and queuing overheads, they do not address the fault tolerance problem.

Machine learning techniques have been used to predict execution time [35]–[37] and system overheads [38], and to develop probability distributions for transient failure characteristics. Duan et.al. [35] used Bayesian network to model and predict workflow task runtimes. The important attributes (e.g. external load, arguments, etc.) are dynamically selected by the Bayesian network and fed into a radial basis function neural network to perform further predictions. Ferreira da Silva et al. [37] used regression trees to dynamically estimate task needs including process I/O, runtime, memory peak, and disk usage. In this work, we use the knowledge obtained in prior works on failure [23], overhead [38], and task runtime analyses [37] as the foundations to build the *prior knowledge* based on the Maximum Likelihood Estimation (MLE) that integrates both the knowledge and runtime feedbacks to adjust the parameter estimation accordingly.

3 Design and Models

3.1 Workflow Management System Model

A workflow is modeled as a directed acyclic graph (DAG), where each node in the DAG often represents a workflow task, and the edges represent dependencies between the tasks that constrain the order in which tasks are executed. *Dependencies* typically represent data-flow dependencies in the application, where the output files produced by one

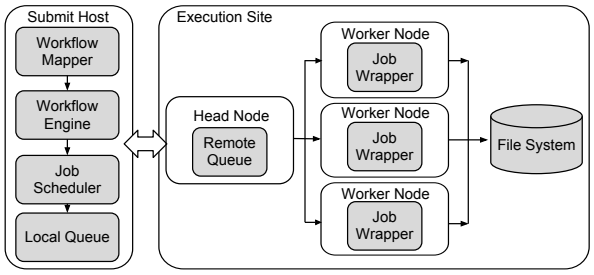


Fig. 1: Overview of the workflow management system.

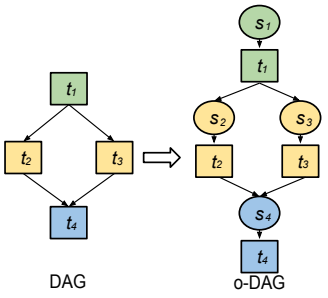


Fig. 2: Extending DAG to o-DAG (s is a system overhead).

task are used as inputs of another task. Each *task* is a computational program and a set of parameters that need to be executed. This model fits several WMS such as Pegasus [22], Askalon [39], and Taverna [40].

In this work, we assume a single execution site with multiple compute resources, such as virtual machines on Cloud platforms. Fig. 1 shows a typical workflow execution environment that targets a homogeneous computer cluster (e.g., a dedicated cluster or a virtual cluster on Clouds). The submit host prepares a workflow for execution (e.g. clustering, mapping, etc.). The jobs are executed remotely on individual worker nodes. The main components are:

Workflow Mapper: Generates an executable workflow from an abstract workflow [22] provided by the user or a workflow composition system. It restructures the workflow to optimize performance, and adds tasks for data management and provenance information generation. In this work, the workflow mapper is also used to merge tasks into a single clustered job (i.e., *task clustering*). A job then is a single execution unit in the workflow execution system and is composed of one or more tasks.

Workflow Engine: Executes jobs defined in the workflow in order of their dependencies. Only jobs that have all their parent jobs completed are submitted to the Job Scheduler. The workflow engine relies on the resources (compute, storage, and network) defined in the executable workflow to perform computations. The time span between the job release and its submission to the Job Scheduler is denoted as the **workflow engine delay**.

Job Scheduler and Local Queue: Manage individual workflow jobs and supervise their execution on local and remote resources. The time span between the job submission to the scheduler and the beginning of its execution on a worker node is denoted as the **queue delay**. This delay reflects both the efficiency of the scheduler and the resource availability.

Job Wrapper: Extracts tasks from clustered jobs and executes them on the worker nodes. The **clustering delay** is the elapsed time of the extraction process.

We extend the DAG model to be overhead aware (o-DAG). System overheads play an important role in workflow execution and constitute a major part of the overall runtime when tasks are poorly clustered [38], in particular for tasks with very short runtimes. Fig. 2 illustrates how we augment a DAG to be an o-DAG with the capability to represent system overheads (s) such as the workflow engine and queue delays. In addition, system overheads also include data transfer delays caused by staging-in and staging-out of data. This classification of system overheads is based on our prior workflow analysis [38]. Table 1 summarizes the system overheads and task runtimes of three real scientific workflows executed on a real distributed platform. Details about these scientific workflow applications will be presented in Section 5.1.

With an o-DAG model, we can explicitly express the process of task clustering. In this work, we address task clustering horizontally and vertically. **Horizontal Clustering** (HC) merges multiple tasks within the same horizontal level of the workflow—the horizontal level of a task is defined as the longest distance from the DAG's entry task to this task. **Vertical Clustering** (VC) merges tasks within a pipeline of the workflow. Tasks in the same pipeline share a *single-parent-single-child* relationship, i.e. a task t_b has a unique parent t_a, which has a unique child t_b.

Fig. 3 shows a simple example on how to perform HC, in which two tasks t_2 and t_3, without data dependency between them, are merged into a clustered job j_1. Job wrappers are often used to execute clustered jobs, but they add an overhead defined as the clustering delay c. The clustering delay measures the difference between the sum of the actual task runtimes and the job runtime seen by the job scheduler. After horizontal clustering, t_2 and t_3 in j_1 can be executed in sequence or in parallel, if parallelism is supported. In a parallel environment, the overall runtime for the workflow in Fig. 3 (left) is $runtime_l = s_1 + t_1 + \max(s_2 + t_2, s_3 + t_3) + s_4 + t_4$, while the overall runtime for the clustered workflow in Fig. 3 (right) is $runtime_r = s_1 + t_1 + s_2 + c_1 + t_2 + t_3 + s_4 + t_4$. $runtime_l > runtime_r$ as long as $c_1 < \max(s_2, s_3)$, which is often the case in many distributed systems since the clustering delay within a single execution node is usually shorter than the scheduling overhead across different execution nodes.

Fig. 4 illustrates an example of vertical clustering, in which tasks t_2, t_4, and t_6 are merged into j_1, while tasks t_3, t_5, and t_7 are merged into j_2. Similarly, clustering delays c_2 and c_3 are added to j_1 and j_2 respectively, but system overheads s_4, s_5, s_6, and s_7 are removed.

In situations where the scheduling and queue overheads are important, the use of task clustering techniques can significantly improve the workflow execution performance. In an ideal scenario, where failures are absent, the number of tasks in a clustered job (clustering size, k) would be

Workflow	Number of Tasks	Number of Nodes	Average Workflow Engine Delay	Average Queue Delay	Average Task Runtime
CyberShake	24142	5	12s	188s	5s
Epigenomics	83	8	6s	311s	158s
SIPHT	33	8	17s	945s	20s

TABLE 1: System overheads and task runtimes gathered from scientific workflow execution traces [38].

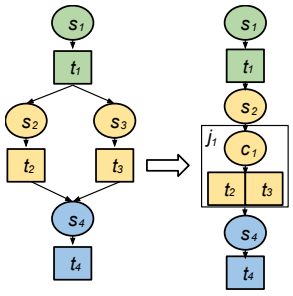


Fig. 3: A simple example of horizontal clustering (color indicates the horizontal level of a task).

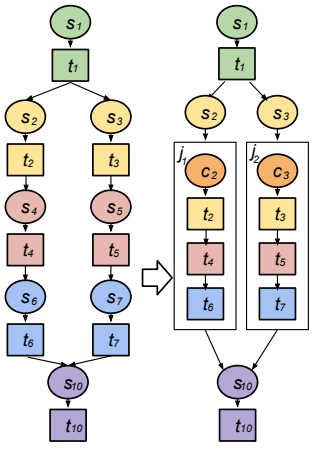


Fig. 4: A simple example of vertical clustering.

defined as the number of all tasks in the queue divided by the number of available resources. Such a naïve setting assures that the number of jobs is equal to the number of resources and the workflow can fully utilize the resources. However, in a faulty environment the clustering size should be defined according to the failure rates, in particular, the task failure rate. Intuitively, if the task failure rate is high, the clustered jobs may need to be re-executed more often compared to the case without clustering. Such performance degradation will counteract the benefits of reducing scheduling overheads. In the rest of this paper, we will show how to adjust k based on the estimated parameters of the task runtime $\boldsymbol{t}$, the system overhead $\boldsymbol{s}$, and the inter-arrival time of task failures $\boldsymbol{\gamma}$.

3.2 Task Failure Model

In our prior work [38], we have verified that system overheads $\boldsymbol{s}$ better fits a Gamma or a Weibull distribution rather than an Exponential or Normal distribution. Schroeder et al. [9] have verified that the inter-arrival time of task failures better fits a Weibull distribution (defined by a shape parameter of 0.78) rather than the Lognormal and Exponential distributions. In [41], transient errors also follow a Weibull distribution. In [42], [43], Weibull, Gamma and Lognormal distributions are among the best fit to estimate task runtimes for a set of workflow traces. Without loss of generality, we choose the Gamma distribution to model the task runtime ($\boldsymbol{t}$) and the system overhead ($\boldsymbol{s}$), and the Weibull distribution to model the inter-arrival time of failures ($\boldsymbol{\gamma}$). $\boldsymbol{s}$, $\boldsymbol{t}$, and $\boldsymbol{\gamma}$ are random variables of all tasks instead of one specific task.

Probability distributions such as Weibull and Gamma are usually described with two parameters: the *shape parameter* (ϕ), and the *scale parameter* (θ). The shape parameter affects the shape of a distribution, for example, whether it is symmetrical or not. The scale parameter affects the stretching or shrinking of a distribution, for example, whether it is approximately uniform or it has a peak. Both parameters control the characteristics of a distribution. For example, the mean of a Gamma distribution is $\phi\theta$ and the Maximum Likelihood Estimation (MLE) is $(\phi-1)\theta$.

Let a, b be the parameters of the prior knowledge, D the observed dataset, and θ the parameter we aim to estimate. In Bayesian probability theory, if the posterior distribution $p(\theta|D,a,b)$ is in the same family as the prior distribution $p(\theta|a,b)$, the prior and the posterior distributions are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function [44]. For instance, the Inverse-Gamma family is conjugate to itself (or self-conjugate) with respect to a Weibull likelihood function: if the likelihood function is Weibull, choosing an Inverse-Gamma prior over the mean will ensure that the posterior distribution is also Inverse-Gamma. Based on this definition, the parameters estimation of our task failure model has its foundations on prior works on failure and performance analyzes [9], [38], [42], [43].

Therefore, once observed data D, the posterior distribution of θ is determined as follows:

$$\begin{aligned} p(\theta|D,a,b) &= \frac{p(\theta|a,b)\times p(D|\theta)}{p(D|a,b)} \\ &\propto p(\theta|a,b)\times p(D|\theta) \end{aligned} \quad (1)$$

where D is the observed inter-arrival time of failures X, the observed task runtime RT, or the observed system overheads S; $p(\theta|D,a,b)$ is the posterior we aim to compute; $p(\theta|a,b)$ is the prior, which we have already known from previous works; and $p(D|\theta)$ is the likelihood. We define $X=\{x_1,x_2,\ldots,x_n\}$ as the observed data of the inter-arrival time of failures $\boldsymbol{\gamma}$ during the execution. Similarly, we define $RT=\{t_1,t_2,\ldots,t_n\}$, and $S=\{s_1,s_2,\ldots,s_n\}$ as the observed data of task runtime $\boldsymbol{t}$ and system overheads $\boldsymbol{s}$ respectively.

More specifically, we model the inter-arrival time of failures ($\boldsymbol{\gamma}$) with a Weibull distribution as in [9], which has a known shape parameter of ϕ_γ and an unknown scale parameter θ_γ: $\boldsymbol{\gamma}\sim W(\theta_\gamma,\phi_\gamma)$.

The conjugate pair of a Weibull distribution with a known shape parameter ϕ_γ is an Inverse-Gamma distribution,

which means if the prior distribution follows an Inverse-Gamma distribution $\Gamma^{-1}(a_\gamma, b_\gamma)$ with the shape parameter as a_γ and the scale parameter as b_γ, then the posterior follows an Inverse-Gamma distribution as follows:

$$\theta_\gamma \sim \Gamma^{-1}(a_\gamma + n, b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}) \tag{2}$$

Equation 2 means the posterior estimation of θ_γ is initially controlled by the prior knowledge (a_γ and b_γ) and then gradually adjusted by observed data (n and x_i). The MLE (Maximum Likelihood Estimation) of the scale parameter θ_γ is defined as:

$$MLE(\theta_\gamma) = \frac{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}{a_\gamma + n + 1} \tag{3}$$

The understanding of the MLE is two fold: in the initial case there is no observed data, thus the MLE is determined by the prior knowledge, i.e. $\frac{b_\gamma}{a_\gamma+1}$; when $n \to \infty$, the MLE $\frac{\sum_{i=1}^{n} x_i^{\phi_\gamma}}{n+1} \to \overline{x^{\phi_\gamma}}$, which means it is determined by the observed data, and is close to the regularized average of the observed data. If the estimation process only utilizes the prior knowledge, it is named **Static Estimation**. If the process utilizes both the prior and posterior knowledge, it is named **Dynamic Estimation**.

We model the task runtime $\boldsymbol{t}$ with a Gamma distribution as in [42], [43], with a known shape parameter ϕ_t and an unknown scale parameter θ_t. The conjugate pair of Gamma distribution with a known shape parameter is also a Gamma distribution. If the prior knowledge follows $\Gamma(a_t, b_t)$, where a_t is the shape parameter and b_t the rate parameter (or $\frac{1}{b_t}$ is the scale parameter), the posterior follows $\Gamma(a_t + n\phi_t, b_t + \sum_{i=1}^{n} t_i)$ with $a_t + n\phi_t$ as the shape parameter and $b_t + \sum_{i=1}^{n} t_i$ as the rate parameter. The MLE of θ_t is then defined as follows:

$$MLE(\theta_t) = \frac{b_t + \sum_{i=1}^{n} t_i}{a_t + n\phi_t - 1} \tag{4}$$

Similarly, if we model the system overhead $\boldsymbol{s}$ with a Gamma distribution with a known shape parameter ϕ_s and an unknown scale parameter θ_s, and the prior knowledge as $\Gamma(a_s, b_s)$, the MLE of θ_s is thus defined as follows:

$$MLE(\theta_s) = \frac{b_s + \sum_{i=1}^{n} s_i}{a_s + n\phi_s - 1} \tag{5}$$

We have assumed that the task runtime, system overheads, and inter-arrival time between failures are a function of task types. The reason is that tasks at different levels (the deepest depth from the entry task to this task) are often of different types in scientific workflows. Given n independent tasks at the same workflow level and the distribution of the task runtime, the system overheads, and the inter-arrival time of failures, we aim at reducing the overall runtime $\boldsymbol{M}$ for completing these tasks by adjusting the clustering size k (the number of tasks in a job). $\boldsymbol{M}$ is also a random variable, which includes the system overheads and the runtime of the clustered job and its subsequent retried jobs if the first attempt fails. We also assume that task failures are independent for each worker node (but with the same distribution) without considering the failures that bring the whole system down (e.g. a failure in the shared file system).

The runtime of a job is a random variable indicated by $\boldsymbol{d}$. A clustered job succeeds only if all of its tasks succeed. The job runtime is the sum of the cumulative task runtime of k tasks and the system overhead. We assume that the task runtime of each task is independent of each other, therefore the cumulative task runtime of k tasks is also a Gamma distribution since the sum of Gamma distributions with the same scale parameter is still a Gamma distribution. We also assume the system overhead is independent of all the task runtimes. A general solution to express the sum of independent Gamma distributions with different scale and shape parameters is provided in [45]. For simplicity, we limit this work to show a typical case where the system overhead and the task runtime have the same scale parameter ($\theta_{ts} = \theta_t = \theta_s$).

Therefore, the job runtime (regardless of whether it succeeds of fails) is defined as follows:

$$\boldsymbol{d} \sim \Gamma(k\phi_t + \phi_s, \theta_{ts}) \tag{6}$$

$$MLE(\boldsymbol{d}) = (k\phi_t + \phi_s - 1)\theta_{ts} \tag{7}$$

Let N be the retry time of clustered jobs. The process to run and retry a job is a Bernoulli trial with exactly two possible outcomes: *success* or *failure*. Once a job fails, it will be re-executed until it is eventually successfully completed (since failures are assumed transient). For a given job runtime d_i, the retry time N_i of a clustered job i is defined as follows:

$$N_i = \frac{1}{1 - F(d_i)} = \frac{1}{e^{-\left(\frac{d_i}{\theta_\gamma}\right)^{\phi_\gamma}}} = e^{\left(\frac{d_i}{\theta_\gamma}\right)^{\phi_\gamma}} \tag{8}$$

where $F(d_i)$ is the CDF (Cumulative Distribution Function) of $\boldsymbol{\gamma}$. The time to complete d_i successfully in a faulty execution environment is determined as follows:

$$M_i = d_i \times N_i = d_i \times e^{\left(\frac{d_i}{\theta_\gamma}\right)^{\phi_\gamma}} \tag{9}$$

Equation 9 has involved two distributions $\boldsymbol{d}$ and θ_γ (ϕ_γ is known). From Equation 2, we have:

$$\frac{1}{\theta_\gamma} \sim \Gamma\left(a_\gamma + n, \frac{1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}\right) \tag{10}$$

$$MLE\left(\frac{1}{\theta_\gamma}\right) = \frac{a_\gamma + n - 1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}} \tag{11}$$

M_i is a monotonic increasing function of both d_i and $\frac{1}{\theta_\gamma}$, and the two random variables are independent of each other, therefore:

$$MLE(M_i) = MLE(d_i) e^{\left(MLE(d_i) MLE\left(\frac{1}{\theta_\gamma}\right)\right)^{\phi_\gamma}} \tag{12}$$

From Equation 12, to attain $MLE(M_i)$ we just need to attain $MLE(d_i)$ and $MLE\left(\frac{1}{\theta_\gamma}\right)$ at the same time. In both

dimensions (d_i and $\frac{1}{\theta_\gamma}$), M_i is a Gamma distribution, and each M_i has the same distribution parameters, therefore:

$$\begin{aligned} \boldsymbol{M} &= \frac{1}{r} \times \sum_{i=1}^{\frac{n}{k}} M_i \sim \Gamma \\ MLE(\boldsymbol{M}) &= \frac{n}{rk} \times MLE(M_i) \\ &= \frac{n}{rk} \times MLE(d_i) e^{\left(MLE(d_i) MLE\left(\frac{1}{\theta_\gamma}\right)\right)^{\phi_\gamma}} \end{aligned} \quad (13)$$

where r is the number of resources. In this work, we consider a compute cluster as a homogeneous cluster, which is often the case in dedicated clusters and cloud platforms.

Let k^* be the optimal clustering size that minimizes Equation 13. $\arg min$ stands for the argument (k) of the minimum [46], i.e., the value of k such that $MLE(\boldsymbol{M})$ attains its minimum value:

$$k^* = \arg min\{MLE(\boldsymbol{M})\} \quad (14)$$

It is not trivial to find an analytical solution of k^*. However, there are a few constraints that can simplify the estimation of k^*: (i) k can only be an integer in practice; (ii) $MLE(\boldsymbol{M})$ is continuous and has one minimum. Methods such as Newton's method can be used to find the minimal $MLE(\boldsymbol{M})$ and the corresponding k. Fig. 5 shows an example of $MLE(\boldsymbol{M})$ using static estimation with a low task failure rate ($\theta_\gamma = 40$s), a medium task failure rate ($\theta_\gamma = 30$s) and a high task failure rate ($\theta_\gamma = 20$s) respectively. Other parameters are $n = 50$, $\phi_t = 5$s and $\phi_s = 50$s, and the scale parameter θ_{ts} is defined as 2 for simplicity. These parameters are close to the parameters of

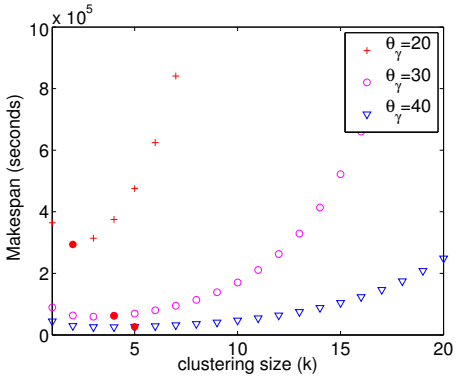


Fig. 5: Makespan with different clustering size and θ_γ. ($n = 1000$, $r = 20$, $\phi_t = 5$s, $\phi_s = 50$s). Red dots are the minimums.

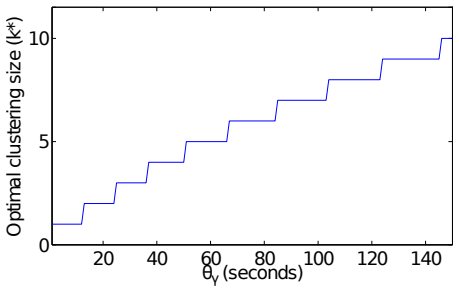


Fig. 6: Optimal clustering size (k^*) with different θ_γ ($n = 1000$, $r = 20$, $\phi_t = 5$s, $\phi_s = 50$s).

tasks at the level of the `mProjectPP` tasks of the Montage workflow (Section 5). Fig. 6 shows the relationship between the optimal clustering size (k^*) and θ_γ, which is a non-decreasing function. The optimal clustering size (red dots in Fig. 5) when $\theta_\gamma \in \{20, 30, 40\}$ is 2, 3, and 5 respectively. As expected, the longer the inter-arrival time of failures is, the lower the task failure rate is. With a lower task failure rate, a larger k value assures that system overheads are reduced without retrying the tasks too many times.

From this theoretic analysis, we conclude that (i) the longer the inter-arrival time of failures is, the better runtime performance the task clustering has; and (ii) by adjusting the clustering size according to the inter-arrival time, the overall runtime performance can be improved.

Parameter	Description
t, s, d	distribution of task runtime, overhead, job runtime
γ	distribution of the inter-arrival time of failures
θ_γ, ϕ_γ	scale and shape parameters of γ
θ_t, ϕ_t	scale and shape parameters of t
θ_s, ϕ_s	scale and shape parameters of s
a_γ, b_γ	prior knowledge of γ
a_t, b_t	prior knowledge of t
a_s, b_s	prior knowledge of s
k	number of tasks in a job
N	distribution of the retry time of clustered jobs
N_i	retry time of job i
M	distribution of the overall runtime
M_i	the overall runtime to complete job i
r	the number of available worker nodes
n	the number of tasks

TABLE 2: Explanation of the symbols used in this work.

4 Fault-Tolerant Clustering

Inappropriate task clustering may negatively impact the workflow makespan in faulty distributed environments. In this section, we propose three fault-tolerant task clustering methods—Selective Reclustering (*SR*), Dynamic Reclustering (*DR*), and Vertical Reclustering (*VR*)—that adjust the clustering size (k) of the jobs to reduce the impact of task failures on the workflow execution. These methods are based on the Horizontal Clustering (HC) [1] technique that has been implemented and used in the Pegasus workflow management system (WMS) [22].

Horizontal Clustering (*HC*). Horizontal clustering merges multiple tasks within the same horizontal level of the workflow. The clustering granularity (number of tasks within a cluster) of a clustered job is controlled by the user, who defines either the number of tasks per clustered job (*clusters.size*), or the number of clustered jobs per horizontal level of the workflow (*clusters.num*). For simplicity, we set *clusters.num* to be the same as the amount of available resources. In [47], we have evaluated the runtime performance for different clustering granularities. Algorithm 1 shows the pseudocode for HC. The `Clustering` and `Merge` procedures are invoked in the initial task clustering process, while the `Reclustering` procedure is invoked when a failed job is detected by the monitoring system. Fig. 7 shows an example for k = 4. As a result, there are four tasks in a clustered job. During execution, three out of these tasks (t_1, t_2, t_3) fail. Due to the lack of an adaptive

Algorithm 1 Horizontal Clustering algorithm.

Require: W: workflow; C: max number of tasks per job defined by *clusters.size* or *clusters.num*

1: **procedure** CLUSTERING(W,C)
2: **for** $level < depth(W)$ **do**
3: $TL \leftarrow$ TASKSATLEVEL(W,$level$) ▷ Divide W based on depth
4: $CL \leftarrow$ MERGE(TL,C) ▷ Returns a list of clustered jobs
5: $W \leftarrow W - TL + CL$ ▷ Merge dependencies as well
6: **end for**
7: **end procedure**
8: **procedure** MERGE(TL,C)
9: $J \leftarrow \{\}$ ▷ An empty job
10: $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
11: **while** TL is not empty **do**
12: J.add (TL.pop(C)) ▷ Pops C tasks that are not merged
13: CL.add(J)
14: **end while**
15: **return** CL
16: **end procedure**
17: **procedure** RECLUSTERING(J) ▷ J is a failed job
18: $J_{new} \leftarrow$ COPYOF(J) ▷ Copy Job J
19: $W \leftarrow W + J_{new}$ ▷ Re-execute it
20: **end procedure**

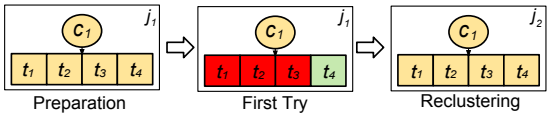


Fig. 7: An example of Horizontal Clustering (red boxes are failed tasks).

mechanism, HC keeps retrying all of the four tasks in the following attempts until all of them succeed.

Selective Reclustering (*SR*). The selective re-clustering technique, on the other hand, merges only failed tasks within a clustered job into a new clustered job. Algorithm 2 shows the pseudocode of the `Reclustering` procedure for the SR method. The `Clustering` and `Merge` procedures are the same as those for HC. Fig. 8 shows an example of the SR method. In the first attempt, the clustered job, composed of 4 tasks, has 3 failed tasks (t_1, t_2, t_3). Three failed tasks are merged into a new clustered job j_2 and retried. This approach does not intend to adjust the clustering size k, although the clustering size may become smaller after each attempt since subsequent clustered jobs may have fewer tasks. In the example, k has decreased from 4 to 3. However, the optimal clustering size may not be 3, which limits the workflow performance if the θ_γ is small and k should be decreased as much as possible. The advantage of SR is that it is simple to implement and can be incorporated into existing WMS with minimum impact on the workflow execution efficiency as shown in Section 5.

Dynamic Reclustering (*DR*). The Selective Reclustering method does not analyze the clustering size, rather it uses a self-adjusted approach to reduce k to the number of failed tasks. However, the actual optimal clustering size may be larger or smaller than the number of failed tasks. Therefore, we propose the Dynamic Reclustering method that merges failed tasks into new clustered jobs in which the clustering size is set to k^* according to Equation 14. Algorithm 3 shows the pseudocode for the `Reclustering` procedure for the DR method. Fig. 9 shows an example where k is

Algorithm 2 Selective Reclustering algorithm.

Require: W: workflow; C: max number of tasks per job defined by *clusters.size* or *clusters.num*

1: **procedure** RECLUSTERING(J) ▷ J is a failed job
2: $TL \leftarrow$ GETTASKS(J)
3: $J_{new} \leftarrow \{\}$ ▷ An empty job
4: **for all** Task t in TL **do**
5: **if** t is failed **then**
6: J_{new}.add (t)
7: **end if**
8: **end for**
9: $W \leftarrow W + J_{new}$ ▷ Re-execute it
10: **end procedure**

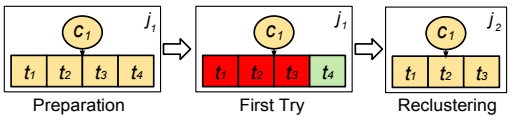


Fig. 8: An example of Selective Reclustering (red boxes are failed tasks). Failed tasks are merged into a new job and retried.

initially set to 4. In the first attempt, 3 tasks within the clustered job have failed. Therefore, there are only 3 tasks to be retried, and thus the clustering size should be decreased to 2 accordingly. Two new clustered jobs j_2 (containing t_1 and t_2) and j_3 (containing t_3) are created. Reducing the granularity of failed clustered jobs may decrease the probability of future failures, since at least one of the jobs will execute on a different worker node.

Vertical Reclustering (*VR*). VR is an extension of the Vertical Clustering (Section 3.1) method. Similar to Selective Reclustering, Vertical Reclustering only retries failed or not

Algorithm 3 Dynamic Reclustering algorithm.

Require: W: workflow; C: max number of tasks per job defined by *clusters.size* or *clusters.num*

1: **procedure** RECLUSTERING(J) ▷ J is a failed job
2: $TL \leftarrow$ GETTASKS(J)
3: $J_{new} \leftarrow \{\}$
4: **for all** Task t in TL **do**
5: **if** t is failed **then**
6: J_{new}.add (t)
7: **end if**
8: **if** $J_{new}.size() > k^*$ **then**
9: $W \leftarrow W + J_{new}$
10: $J_{new} \leftarrow \{\}$
11: **end if**
12: **end for**
13: $W \leftarrow W + J_{new}$ ▷ Re-execute it
14: **end procedure**

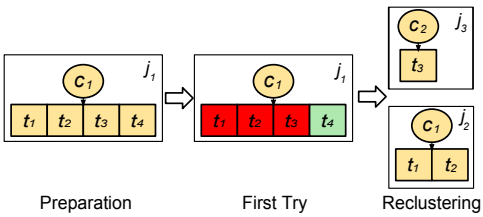


Fig. 9: An example of Dynamic Reclustering (red boxes are failed tasks). The clustering size k is adjusted to 2 and thus failed tasks are merged into two new clustered jobs.

Algorithm 4 Vertical Reclustering algorithm.

Require: W: workflow;
1: **procedure** CLUSTERING(W)
2: **for** $level < depth(W)$ **do**
3: $TL \leftarrow$ TASKSATLEVEL($W, level$) ▷ Divide W based on depth
4: $CL, TL_{merged} \leftarrow$ MERGE(TL) ▷ List of clustered jobs
5: $W \leftarrow W - TL_{merged} + CL$ ▷ Merge dependencies as well
6: **end for**
7: **end procedure**
8: **procedure** MERGE(TL)
9: $TL_{merged} \leftarrow TL$ ▷ All the tasks that have been merged
10: $CL \leftarrow \{\}$ ▷ An empty list of clustered jobs
11: **for all** t in TL **do**
12: $J \leftarrow \{t\}$
13: **while** t has only 1 child t_{child} and t_{child} has only 1 parent **do**
14: J.add (t_{child})
15: $TL_{merged} \leftarrow TL_{merged} + t_{child}$
16: $t \leftarrow t_{child}$
17: **end while**
18: CL.add(J)
19: **end for**
20: **return** CL, TL_{merged}
21: **end procedure**
22: **procedure** RECLUSTERING(J) ▷ J is a failed job
23: $TL \leftarrow$ GETTASKS(J)
24: $k^* \leftarrow J.size()/2$ ▷ Reduce the clustering size by half
25: $J_{new} \leftarrow \{\}$
26: **for all** Task t in TL **do**
27: **if** t is failed or not completed **then**
28: J_{new}.add (t)
29: **end if**
30: **if** $J_{new}.size() > k^*$ **then**
31: $W \leftarrow W + J_{new}$; $J_{new} \leftarrow \{\}$
32: **end if**
33: **end for**
34: $W \leftarrow W + J_{new}$ ▷ Re-execute it
35: **end procedure**

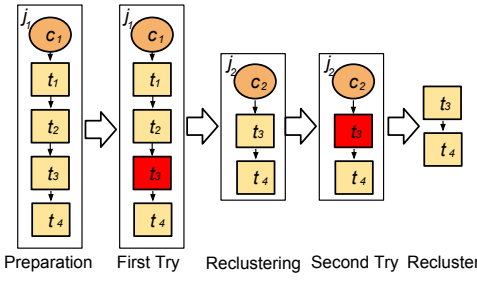


Fig. 10: An example of Vertical Reclustering (red boxes are failed tasks). The clustering size is decreased by half when a job execution fails.

completed tasks. If a failure is detected, k is decreased by half and failed tasks are re-clustered accordingly. Fig. 10 shows an example of VR where tasks within a pipeline are initially merged into a single clustered job (t_1, t_2, t_3, t_4). t_3 fails at the first attempt assuming it is a failure-prone task (i.e., its θ_γ is short). VR then retries only the failed task (t_3) and tasks that have not been yet completed (t_4) by merging them into a new job j_2. In the second attempt, j_2 fails and then it is divided into two single task jobs (t_3 and t_4). Since the clustering size is minimum ($k = 1$), VR performs no vertical clustering and continue retrying t_3 and t_4 (but still following their data dependency) until they succeed. Algorithm 4 shows the pseudocode for the VR method.

5 Experiments and Discussions

In this section, we evaluate our methods with five scientific workflow applications, whose runtime information is gathered from real execution traces. We conduct a simulation-based approach in which we vary system parameters such as the task failures inter-arrival time in order to evaluate the reliability of our fault-tolerant task clustering methods.

5.1 Scientific Workflow Applications

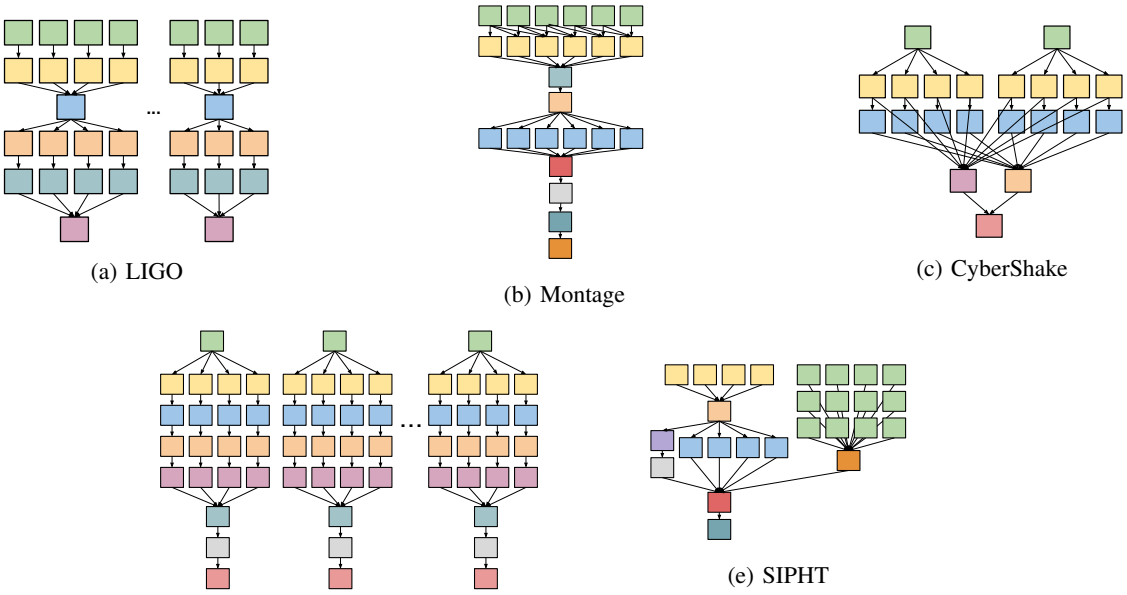
In the experiments, we use the following scientific workflow applications: LIGO Inspiral analysis, Montage, CyberShake, Epigenomics, and SIPHT. Below, we briefly describe each of them and present their main characteristics and structures:

LIGO. Laser Interferometer Gravitational Wave Observatory (LIGO) [48] workflows are used to search for gravitational wave signatures in data collected by large-scale interferometers. The observatories' mission is to detect and measure gravitational waves predicted by general relativity (Einstein's theory of gravity), in which gravity is described as due to the curvature of the fabric of time and space. Fig. 11a shows a simplified version of the workflow. The LIGO Inspiral workflow is separated into multiple groups of interconnected tasks, which we call branches in the rest of this work. Each branch may have a different number of pipelines. The LIGO workflow is a data-intensive workflow.

Montage. Montage [49] is an astronomy application used to construct large image mosaics of the sky. Input images are reprojected onto a sphere and overlap is calculated for each input image. The application re-projects input images to the correct orientation while keeping background emission level constant in all images. Images are added by rectifying them to a common flux scale and background level. Finally the reprojected images are co-added into a final mosaic. The resulting mosaic image can provide a much deeper and detailed understanding of the portion of the sky in question. Fig. 11b illustrates a small Montage workflow. The size of the workflow depends on the number of images used in constructing the desired mosaic of the sky.

Cybershake. CyberShake [50] is a seismology application that calculates probabilistic seismic hazard curves for geographic sites in the Southern California region. It identifies all ruptures within 200km of the site of interest and converts rupture definition into multiple rupture variations with differing hypocenter locations and slip distributions. It calculates synthetic seismograms for each rupture variance from where peak intensity measures are extracted and combined with the original rupture probabilities to produce probabilistic seismic hazard curves for the site. Fig. 11c shows an illustration of the Cybershake workflow.

Epigenomics. The Epigenomics workflow [51] is a CPU-intensive application. Initial data are acquired from the Illumina-Solexa Genetic Analyzer in the form of DNA sequence lanes. Each Solexa machine can generate multiple lanes of DNA sequences. These data are converted into a



(a) LIGO (b) Montage (c) CyberShake

(d) Epigenomics (multiple branches) (e) SIPHT

Fig. 11: Simplified visualization of the scientific workflows.

format that can be used by sequence mapping software. The workflow maps DNA sequences to the correct locations in a reference Genome. Then it generates a map that displays the sequence density showing how many times a certain sequence expresses itself on a particular location on the reference genome. The simplified structure of Epigenomics is shown in Fig. 11d.

SIPHT. The SIPHT workflow [52] conducts a wide search for small untranslated RNAs (sRNAs) that regulates several processes such as secretion or virulence in bacteria. The kingdom-wide prediction and annotation of sRNA encoding genes involves a variety of individual programs that are executed in the proper order using Pegasus [22]. These involve the prediction of ρ-independent transcriptional terminators, BLAST (Basic Local Alignment Search Tools [53]) comparisons of the inter genetic regions of different replicons and the annotations of any sRNAs that are found. A simplified structure of the SIPHT workflow is shown in Fig. 11e.

Table 3 shows the summary of the main *workflow characteristics*: number of tasks, average data size, and average task runtimes for the five workflows.

	# Tasks	Avg. Data Size	Avg. Task Runtime
LIGO	800	5 MB	228s
Montage	300	3 MB	11s
CyberShake	700	148 MB	23s
Epigenomics	165	355 MB	2952s
SIPHT	1000	360 KB	180s

TABLE 3: Summary of the main workflow characteristics.

5.2 Experiment Conditions

We adopt a trace-based simulation approach, where we extend our WorkflowSim [54] simulator with the fault-tolerant clustering methods to simulate a controlled distributed environment, where system (i.e., the inter-arrival time of failures) and workflow (i.e., avg. task runtime) settings can be varied to fully explore the performance of our fault-tolerant clustering algorithms. WorkflowSim is an open source workflow simulator that extends CloudSim [55] by providing support for task clustering, task scheduling, and resource provisioning at the workflow level. It has been recently used in multiple workflow study areas [2], [54], [56] and its correctness has been verified in [54].

The simulated computing platform is composed of 20 single homogeneous core virtual machines (worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [57] and FutureGrid [58]. This assumption is also consistent with the setting of many real execution experiments [59]. Each simulated virtual machine (VM) has 512MB of memory and the capacity to process 1,000 million instructions per second. The default network bandwidth is 15MB according to the execution environment in FutureGrid, where our traces were collected. In our execution model, the network bandwidth is the maximum allowed data transfer speed between a pair of virtual machines per file. By default, tasks at the same horizontal level are merged into 20 clustered jobs, which is a simple granularity control selection based on the strength of task clustering (as shown in [47]).

Workload Dataset. We collected workflow execution traces [38], [59] (including overhead and task runtime information) from real runs of the 5 scientific workflow applications previously described. The traces are used to feed the Workflow Generator [60] toolkit to create synthetic workflows. The toolkit uses statistical data gathered from traces of actual scientific workflow executions to generate realistic, synthetic workflows that resemble the real applications. The number of inputs to be processed, the number of tasks in the workflow, and their composition determine the

structure of the generated workflow. Traces, profile data, and characterizations are freely available online for the community [61]. For the experiments a synthetic workflow was generated for each workflow application according to the characteristics shown in Table 3.

Experiment Sets. Three sets of experiments are conducted. Experiment 1 evaluates the performance of our fault-tolerant clustering methods (DR, VR, and SR) over an existing task clustering method (HC), which is devoided of fault-tolerant mechanisms. The goal of the experiment is to identify conditions where each method works best and worst. We also evaluate the performance improvement for different θ_γ values (the scale parameter of the distribution of the inter-arrival time of task failures). θ_γ ranges from 1 to 10 times of the average task runtime such that the workflows run in a reasonable amount of time and the performance difference is visually explicit.

Experiment 2 evaluates the performance impact of the variation of the average task runtime per level (defined as the average task runtime of all the tasks per level), and the average system overheads per level for one scientific workflow application (CyberShake). In particular, we are interested in the performance of DR based on the results of Experiment 1. For this experiment, we define $\theta_\gamma = 100$ since it better highlights the difference between the four methods. We vary the average task runtime of the CyberShake workflow (originally of about 23 seconds, Table 3) by using a multiplier factor $f_r \in [0.5, 1.3]$. We also vary the average system overheads (originally of about 50 seconds) by using a multiplier factor $f_o \in [0.2, 1.8]$.

Experiment 3 evaluates the performance of the static and dynamic estimation. In the static estimation process, only the prior knowledge (shape and scale parameters of a Inverse-Gamma distribution for the inter-arrival time of failures, and a Gamma distribution for the task runtime and system overhead) is used to estimate the MLEs of the unknown parameters (scales parameters of the inter-arrival failures θ_γ, the task runtime θ_t, and the system overhead θ_s). In contrast, the dynamic estimation process leverages the runtime data collected during the execution and update the MLEs respectively. In this experiment, the prior knowledge such as the shape parameter a_γ, and the scale parameter b_γ of the inter-arrival time of failures θ_γ are set manually based on our experience and other researchers' work [9]. The runtime data, such as the series of the actual inter-arrival time of tasks x_i, are collected during the simulation execution and used to adjust all the MLEs based on Equations 3, 4, 5, 7, 11.

Inter-arrival Time of Failures Function ($\theta_\gamma(t)$). The experiments use two sets of the θ_γ function. The first is a step function (Fig. 12), in which θ_γ is decreased from 500 seconds to 50 seconds at time T_d. The step function of θ_γ at time t_c is defined as follows:

$$\theta_\gamma(t_c) = \begin{cases} 50 & \text{if } t_c \geq T_d \\ 500 & \text{if } 0 < t_c < T_d \end{cases} \quad (15)$$

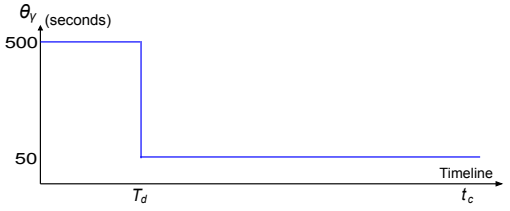


Fig. 12: A Step Function of θ_γ. t_c is the current time and T_d is the moment θ_γ changes from 500 to 50 seconds.

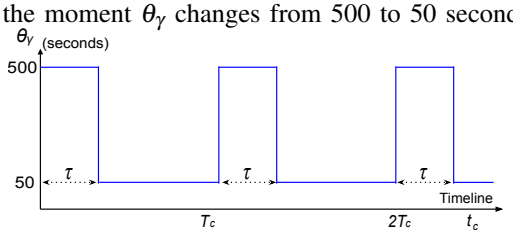


Fig. 13: A Pulse Function of θ_γ. t_c is the current time and T_c is the period of the wave. τ is the width of the pulse.

This function simulates a scenario, where failures happen more frequently than expected. We evaluate the performance difference of dynamic and static estimations for $1000 \leq T_d \leq 5000$ based on the estimation of the workflow makespan. Theoretically, the later we change θ_γ, the less the re-clustering is influenced by the estimation error, and thus the smaller the workflow makespan is. There is one special case when $T_d \to 0$, which means the prior knowledge is wrong at the very beginning.

The second function is a pulse wave function (Fig. 13) in which the amplitude alternates at a steady frequency between a fixed minimum (50 seconds) to a maximum (500 seconds) value. The function is defined as follows:

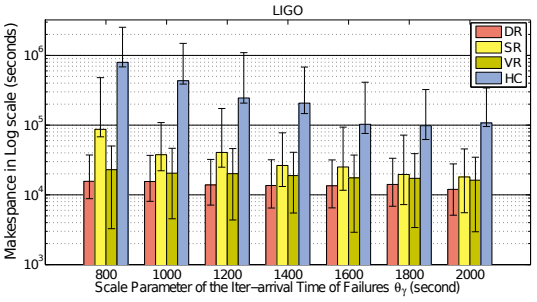
$$\theta_\gamma(t_c) = \begin{cases} 500 & \text{if } 0 < t_c \leq \tau \\ 50 & \text{if } \tau < t_c < T_c \end{cases} \quad (16)$$

where T_c is the period, and τ is the duty cycle of the oscillator. This function simulates a scenario where the failures follow a periodic pattern [62] extracted from failure traces obtained from production distributed systems. In this work, we vary T_c from 1,000 seconds to 10,000 seconds based on the estimation of the workflow's makespan, and τ from $0.1T_c$ to $0.5T_c$.

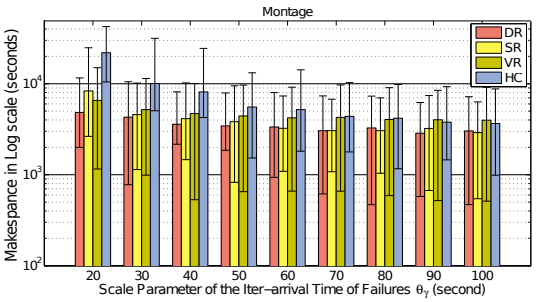
5.3 Results and Discussion

Experiment 1. Fig. 14 shows the performance of the Horizontal Clustering (HC), Selective Reclustering (SR), Dynamic Reclustering (DR), and Vertical Reclustering (VR) methods for the five workflows. DR, SR, and VR significantly improve the makespan when compared to HC in a large scale. By decreasing the inter-arrival time of failures (θ_γ) and thereby generating more task failures, the performance improvement of our methods becomes more significant. Among the three methods, DR and VR perform consistently better than SR, which fails to improve the overall makespan when θ_γ is small. The reason is that SR does not adjust k according to the occurrence of failures.

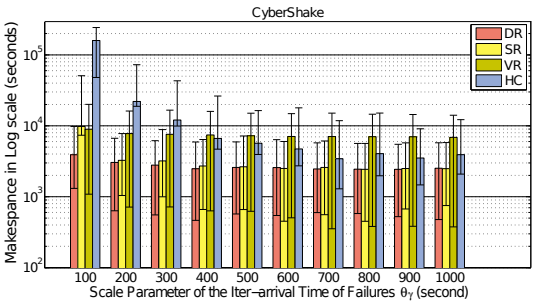
The performance of VR is tightly coupled to the workflow structure and the average task runtime. For example,



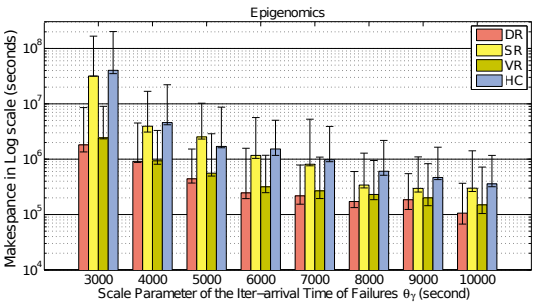
(a) LIGO workflow



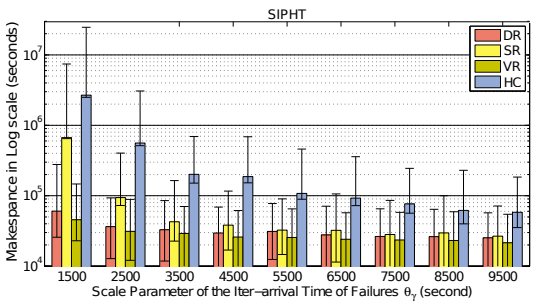
(b) Montage workflow



(c) CyberShake workflow



(d) Epigenomics workflow



(e) SIPHT workflow

Fig. 14: (*Experiment 1*): Performance evaluation of our fault-tolerant task clustering methods for different values of the inter-arrival time (θ_γ).

according to Fig. 11d and Table 3, the Epigenomics workflow has a long task runtime (around 50 minutes) and the pipeline length is 4. This means that vertical clustering creates very long jobs ($\sim 50 \times 4 = 200$ minutes) and thereby VR is more sensitive to the decrease of γ. As indicated in Fig. 14.d, the makespan increases more significantly with the decrease of θ_γ than for other workflows. In contrast, vertical clustering does not improve makespan in the CyberShake workflow (Fig. 14.c) since it does not have many pipelines (Fig. 11c). In addition, the average task runtime of the CyberShake workflow is relatively short (around 23 seconds). Compared to horizontal clustering methods such as HC, SR, and DR, vertical clustering does not generate long jobs and thus the performance of VR is less sensitive to the variation of the scale parameter of the distribution of the inter-arrival time of failures θ_γ.

Table 4 shows the average number of tasks for the minimum ($\min(\theta_\gamma)$) and maximum ($\max(\theta_\gamma)$) values of the scale parameter of the inter-arrival time of failures. Most of the algorithms have same performance when the inter-arrival time of failures is sparse, except for HC. For small θ_γ values, DR significantly reduces the number of tasks.

Experiment 2. Fig. 15 shows the performance (CyberShake workflow) of the task clustering methods, when the task runtime is varied by a multiplier factor f_r. By increasing the task runtime, HC has the most negative impact on the workflow's makespan (increase from a scale of 10^4 to $\sim 10^6$). This is due to the lack of fault-tolerant mechanisms, since HC retries the entire clustered job. SR and VR, however, retry only failed tasks that are merged into a new clustered job. Although the selective and vertical methods

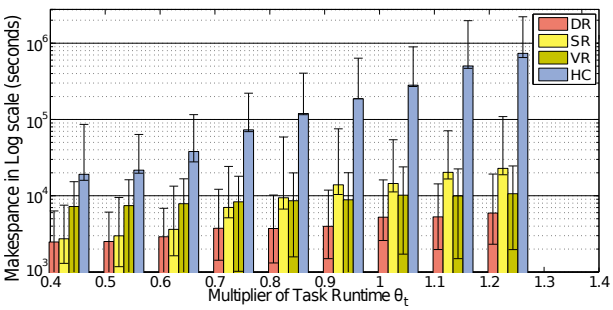


Fig. 15: (*Experiment 2*): Influence of varying task runtime (θ_t) on makespan for the CyberShake workflow.

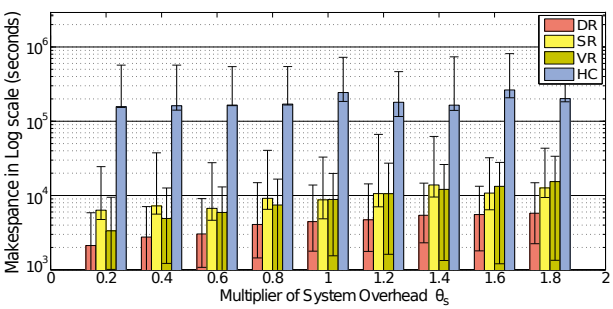


Fig. 16: (*Experiment 2*): Influence of varying system overhead (θ_s) on makespan for the CyberShake workflow.

Workflow	HC		VR		SR		DR	
	$\min(\theta_\gamma)$	$\max(\theta_\gamma)$	$\min(\theta_\gamma)$	$\max(\theta_\gamma)$	$\min(\theta_\gamma)$	$\max(\theta_\gamma)$	$\min(\theta_\gamma)$	$\max(\theta_\gamma)$
LIGO	9765	2300	1113	966	1569	924	939	848
Montage	2467	613	469	331	694	333	411	335
CyberShake	30470	1014	849	714	1480	712	786	712
Epigenomics	6134	369	1016	311	3304	335	455	202
SIPHT	18803	1693	1158	1016	1640	1015	1113	1018

TABLE 4: (*Experiment 1*): Average number of tasks for min and max values of θ_γ for each algorithm.

significantly speedup the execution when compared to HC, they have no mechanism to adjust the clustering size to the actual optimal clustering size k, which may be larger or smaller than the number of failed tasks. By dynamically adjusting the clustering size, DR yields better makespans, in particular for high values of θ_t.

Fig. 16 shows the performance (for the CyberShake workflow) of the task clustering methods when the system overhead is varied by using a multiplier factor f_o. Similarly, with the increase of the system overhead, HC is significantly impacted while SR and VR perform better. Again, for high θ_s values DR performs best. Note that the improvement gained by the fault-tolerant methods is less significant than the performance improvements shown in Fig. 15. The reason is that clustered jobs may have multiple tasks but only one system overhead per job.

Experiment 3. Fig. 17 shows the performance evaluation of the dynamic and static estimations for the CyberShake workflow with a step function of θ_γ. In this experiment, we use DR as the fault-tolerant task clustering method since it yielded the best performance in the past experiments. The step signal function changes the inter-arrival time of failures (θ_γ) from 500 to 50 seconds at time T_d. For high values of T_d, failures are scarce since θ_γ is close to the workflow makespan. Therefore, the influence of θ_γ is negligible, and thus both the static and dynamic estimations have the same behavior. When failures are more frequent, i.e., low values for T_d, the dynamic estimation improves the workflow's makespan by up to 22%. This improvement is due to the ability of the dynamic estimation process to update the MLEs of θ_γ and adapt the clustering size.

Fig. 18 shows the performance evaluation of the dynamic and static estimations with a pulse function of θ_γ. For this experiment, we define the width of the pulse $\tau = 0.1 \cdot T_c$, $0.3 \cdot T_c$, $0.5 \cdot T_c$, where T_c is the period of the wave (Equation 16). For $\tau = 0.1 \cdot T_c$ (Fig. 18.a), the dynamic estimation

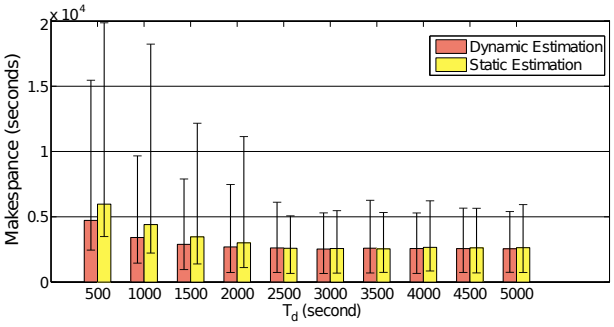
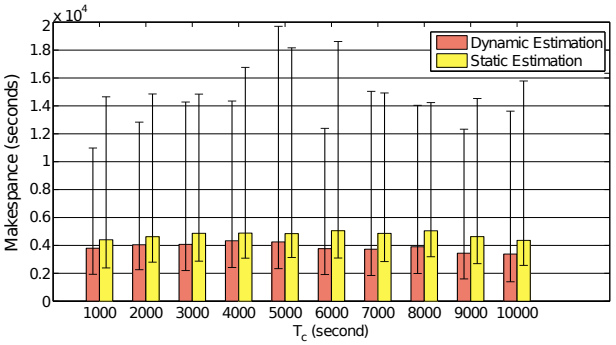
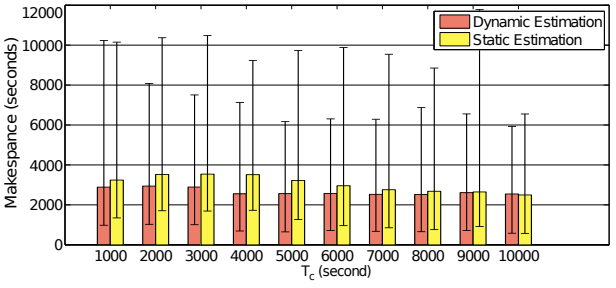


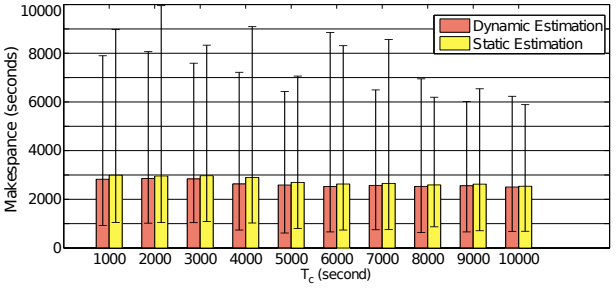
Fig. 17: (*Experiment 3*): Performance evaluation of the static and dynamic estimations for the CyberShake workflow using a step function.



(a) Pulse $\tau = 0.1 \cdot T_c$



(b) Pulse $\tau = 0.3 \cdot T_c$



(c) Pulse $\tau = 0.5 \cdot T_c$

Fig. 18: (*Experiment 3*): Performance evaluation of the static and dynamic estimations for the CyberShake workflow using a pulse function.

improves the makespan by up to 25.7% when compared to the static estimation case. For $\tau = 0.3 \cdot T_c$ (Fig. 18.b), the performance gain of dynamic estimation over static estimation is up to 27.3%. For $T_c = 1000$, the performance gain is not significant since the inter-arrival time of failures changes frequently and then the dynamic estimation process is not able to update swiftly. For $T_c = 10000$, the performance difference is negligible since the inter-arrival time is close to the workflow makespan. For $\tau = 0.5 \cdot T_c$, the gain of dynamic over static estimation is negligible and nearly constant (up to 9.1%) since θ_γ has equal influence on the failure occurrence regardless of its value.

Our experimental results show that adaptive task reclustering methods results in better performance than simple static methods that retry failed clustered jobs. However,

the performance gain may be significantly impacted if the job clustering size is not adjusted to the nearly optimal size. The Dynamic Reclustering algorithm outperforms most of the methods for the 5 workflow applications, however, it yields poorer performance when the workflow structure is irregular (e.g., SIPHT workflow). In this case, Vertical Reclustering would be more suitable.

6 Conclusion and Future Work

In this work, we modeled transient failures in a distributed environment and assess their influence on task clustering. We proposed three dynamic clustering methods to improve the fault tolerance of task clustering and applied them to five widely used scientific workflows. Experimental results showed that the proposed methods significantly improve the workflow's makespan when compared to an existing task clustering method used in workflow management systems. In particular, the Dynamic Reclustering method performed best among all methods since it could adjust the clustering size based on the Maximum Likelihood Estimation of task runtime, system overheads, and the inter-arrival time of failures. The Vertical Reclustering method significantly improved the performance for workflows that had short task runtimes. The dynamic estimation process, which used data collected during the workflow execution, could further improve the overall runtime in a dynamic environment where the inter-arrival time of failures fluctuated.

This work focused on the evaluation of fault-tolerant task clustering techniques on homogeneous environments. In the future, we plan to combine our work with fault-tolerant scheduling in heterogeneous environments, i.e, a scheduling algorithm that avoids mapping clustered jobs to failure-prone nodes. We also intend to combine vertical clustering methods with horizontal clustering methods. For example, vertical clustering can be performed either before or after horizontal clustering, which we believe would bring different performance improvement.

We assumed that the inter-arrival time of transient failures is a function of task type, which is one of the major impact factors. In the future, we plan to consider other factors such as the execution site, which may improve the accuracy of the model. In this paper we assumed that the network bandwidth is the maximum possible data transfer speed between a pair of virtual machines per file. Future work will consider different network models to explore their impact on our fault-tolerant clustering techniques.

Acknowledgments

This work was supported by NFS under grant number IIS-0905032, and SI^2–SSI program, award number ACI–1148515. We thank Boleslaw Szymanski, Gideon Juve, Karan Vahi, Mats Rynge, and Rajiv Mayani for their valuable help. Traces are collected from experiments conducted on FutureGrid, which is supported by NSF under grant FutureGrid 0910812.

References

[1] G. Singh, M. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, G. Mehta, Workflow task clustering for best effort systems with pegasus, in: 15th ACM Mardi Gras Conference, 2008.

[2] W. Chen, E. Deelman, Fault tolerant clustering in scientific workflows, in: IEEE Eighth World Congress on Services, 2012, pp. 9–16.

[3] R. Ferreira da Silva, T. Glatard, F. Desprez, On-line, non-clairvoyant optimization of workflow activity granularity on grids, in: Euro-Par 2013 Parallel Processing, Vol. 8097 of LNCS, 2013, pp. 255–266.

[4] K. Maheshwari, et al., Job and data clustering for aggregate use of multiple production cyberinfrastructures, in: 5th Inter. workshop on Data-Intensive Distributed Computing, 2012.

[5] R. Ferreira da Silva, T. Glatard, F. Desprez, Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions, Concurrency and Computation: Practice and Experience 26 (14) (2014) 2347–2366. doi:10.1002/cpe.3303.

[6] W. Chen, E. Deelman, Integration of workflow partitioning and resource provisioning, in: The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12), 2012.

[7] Y. Zhang, M. S. Squillante, Performance implications of failures in large-scale cluster scheduling, in: The 10th Workshop on Job Scheduling Strategies for Parallel Processing, 2004.

[8] D. Tang, et al., Failure analysis and modeling of a vaxcluster system, in: Int. Symp. on Fault-tolerant computing, 1990.

[9] B. Schroeder, G. A. Gibson, A large-scale study of failures in high-performance computing systems, in: Int. Conf. on Dependable Systems and Networks, 2006.

[10] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, Y. Zhang, Failure data analysis of a large-scale heterogeneous server environment, in: International Conf. on Dependable Systems and Networks, 2004.

[11] J. Bresnahan, T. Freeman, et al., Managing appliance launches in infrastructure clouds, in: Teragrid Conference, 2011.

[12] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, M. Livny, Pegasus: Mapping scientific workflows onto the grid, in: Across Grid Conference, 2004.

[13] R. Duan, et al., Run-time optimisation of grid workflow applications, in: 7th IEEE/ACM Inter. Conf. on Grid Computing, 2006, pp. 33–40.

[14] R. Ferreira da Silva, T. Glatard, A science-gateway workload archive to study pilot jobs, user activity, bag of tasks, task sub-steps, and workflow executions, in: Euro-Par 2012: Parallel Processing Workshops, Vol. 7640, 2013. doi:10.1007/978-3-642-36949-0_10.

[15] E. Deelman, G. Singh, M. Livny, B. Berriman, J. Good, The cost of doing science on the cloud: The montage example, in: 2008 ACM/IEEE Conference on Supercomputing, 2008.

[16] G. B. Berriman, G. Juve, E. Deelman, et al., The application of cloud computing to astronomy: A study of cost and performance, in: Workshop on e-Science challenges in Astronomy and Astrophysics, 2010.

[17] Y. Zhang, A. Mandal, C. Koelbel, K. Cooper, Combined fault tolerance and scheduling techniques for workflow applications on computational grids, in: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009, pp. 244–251.

[18] J. Montagnat, et al., Workflow-based comparison of two distributed computing infrastructures, in: 5th Workshop on Workflows in Support of Large-Scale Science, 2010, pp. 1–10.

[19] G. Kandaswamy, A. Mandal, D. Reed, Fault tolerance and recovery of scientific workflows on computational grids, in: 8th IEEE Inter. Symp. on Cluster Computing and the Grid, 2008, 2008, pp. 777–782.

[20] K. Plankensteiner, et al., A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact, in: 5th IEEE Inter. Conf. on e-Science, 2009, pp. 313–320.

[21] D. Oppenheimer, A. Ganapathi, D. A. Patterson, Why do internet services fail and what can be done about it?, Computer Science Division, University of California, 2002.

[22] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systemsdoi:10.1016/j.future.2014.10.008.

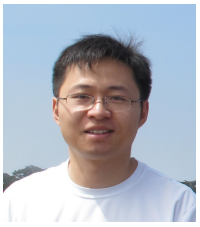
[23] T. Samak, D. Gunter, M. Goode, E. Deelman, G. Mehta, F. Silva, K. Vahi, Failure prediction and localization in large scientific workflows, in: The 6th Workshop on Workflows in Suppporting of Large-Scale Science, 2011.

[24] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertész, P. Kacsuk, Fault detection, prevention and recovery in current grid workflow systems, in: Grid and Services Evolution, 2009, pp. 1–13.

[25] N. Muthuvelu, et al., A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids, in: Australasian workshop on Grid computing and e-research, 2005.

[26] N. Muthuvelu, I. Chai, C. Eswaran, An adaptive and parameterized job grouping algorithm for scheduling grid jobs, in: 10th Inter. Conf. on Advanced Communication Technology, 2008, pp. 975 –980.

[27] N. Muthuvelu, I. Chai, E. Chikkannan, R. Buyya, On-line task granularity adaptation for dynamic grid applications, in: Algorithms and Architectures for Parallel Processing, Vol. 6081 of LNCS, 2010.
[28] W. K. Ng, T. Ang, T. Ling, C. Liew, Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing, Malaysian Journal of Computer Science 19 (2) (2006) 117–126.
[29] T. Ang, W. Ng, T. Ling, L. Por, C. Liew, A bandwidth-aware job grouping-based scheduling on grid environment, Information Technology Journal 8 (2009) 372–377.
[30] Q. Liu, Y. Liao, Grouping-based fine-grained job scheduling in grid computing, in: First International Workshop on Education Technology and Computer Science, 2009.
[31] H. Topcuoglu, et al., Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.
[32] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, K. Kennedy, Task scheduling strategies for workflow-based applications in grids, in: 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05), 2005.
[33] M. Wieczorek, R. Prodan, T. Fahringer, Scheduling of scientific workflows in the askalon grid environment, in: ACM SIGMOD Record, Vol. 34, 2005, pp. 56–62.
[34] S. Kalayci, et al., Distributed and adaptive execution of condor dagman workflows, in: 22nd International Conference on Software Engineering and Knowledge Engineering, 2010.
[35] R. Duan, et al., A hybrid intelligent method for performance modeling and prediction of workflow activities in grids, in: 9th IEEE/ACM Inter. Symp. on Cluster Computing and the Grid, 2009, pp. 339–347.
[36] H. Li, D. Groep, L. Wolters, Efficient response time predictions by exploiting application and resource state similarities, in: The 6th IEEE/ACM International Workshop on Grid Computing, 2005, p. 8.
[37] R. Ferreira da Silva, G. Juve, E. Deelman, et al., Toward fine-grained online task characteristics estimation in scientific workflows, in: 8th Workshop on Workflows in Support of Large-Scale Science, 2013, pp. 58–67. doi:10.1145/2534248.2534254.
[38] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: 6th Workshop on Workflows in Support of Large-Scale Science, 2011.
[39] T. Fahringer, et al., Askalon: A development and grid computing environment for scientific workflows, in: Workflows for e-Science, 2007, pp. 450–471.
[40] T. Oinn, et al., Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.
[41] S. R. McConnel, D. P. Siewiorek, M. M. Tsao, The measurement and analysis of transient errors in digital computer systems, in: Proc. 9th Int. Symp. Fault-Tolerant Computing, 1979, pp. 67–70.
[42] X.-H. Sun, M. Wu, Grid harvest service: a system for long-term, application-level task scheduling, in: International Parallel and Distributed Processing Symposium (IPDPS), 2003, p. 8.
[43] A. Iosup, O. Sonmez, S. Anoep, D. Epema, The performance of bags-of-tasks in large-scale distributed systems, in: 17th Inter. Symp. on High Performance Distributed Computing, 2008, pp. 97–108.
[44] P. Diaconis, D. Ylvisaker, et al., Conjugate priors for exponential families, The Annals of statistics 7 (2) (1979) 269–281.
[45] S. Nadarajah, A review of results on sums of random variables, Acta Applicandae Mathematicae 103 (2) (2008) 131–140.
[46] Arg max, http://en.wikipedia.org/wiki/Arg_max.
[47] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Balanced task clustering in scientific workflows, in: IEEE 9th Inter. Conf. on eScience, 2013, pp. 188–195. doi:10.1109/eScience.2013.40.
[48] LIGO, http://www.ligo.caltech.edu.
[49] G. B. Berriman, E. Deelman, et al., Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in: SPIE Conference on Astronomical Telescopes and Instrumentation, 2004.
[50] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, et al., Cybershake: A physics-based seismic hazard model for southern california, Pure and Applied Geophysics 168 (3-4) (2010) 367–381.
[51] USC Epigenome Center, http://epigenome.usc.edu.
[52] SIPHT, http://pegasus.isi.edu/applications/sipht.
[53] BLAST, http://blast.ncbi.nlm.nih.gov/Blast.cgi.
[54] W. Chen, E. Deelman, Workflowsim: A toolkit for simulating scientific workflows in distributed environments, in: The 8th IEEE International Conference on eScience, 2012.
[55] R. N. Calheiros, et al., CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1).
[56] F. Jrad, J. Tao, A. Streit, A broker-based framework for multi-cloud workflows, in: 2013 international workshop on Multi-cloud applications and federated clouds, 2013, pp. 61–68.
[57] Amazon Web Services, http://aws.amazon.com.
[58] FutureGrid, http://futuregrid.org/.
[59] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Generation Computer Systems 29 (3) (2013) 682 – 692.
[60] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling and evaluating research on scientific workflows, in: 10th IEEE International Conference on e-Science, 2014, pp. 177–184. doi:10.1109/eScience.2014.44.
[61] Workflow Archive, http://workflowarchive.org.
[62] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, D. Epema, Analysis and modeling of time-correlated failures in large-scale distributed systems, in: 11th Inter. Conf. on Grid Computing, 2010, pp. 65–72.



Weiwei Chen received his PhD in Computer Science from University of Southern California, USA in 2014. In 2009, he completed his bachelor in the Dept. of Automation, Tsinghua University, China. His research interests include distributed computing, service computing and data analysis. He currently works at Google's infrastructure team.

Rafael Ferreira da Silva is a Computer Scientist at the USC Information Sciences Institute. He received his PhD in Computer Science from INSA-Lyon, France, in 2013. His research focuses on the optimization of the execution of scientific workflows on heterogeneous distributed systems. See http://www.rafaelsilva.com for further information.

Ewa Deelman is a Research Associate Professor at the USC Computer Science Department and a Assistant Division Director at the USC Information Sciences Institute. Her research interests include the design and exploration of distributed scientific environments, with emphasis on workflow management. She received her PhD in Computer Science from the Rensselaer Polytechnic Institute in 1997.

Thomas Fahringer received the Ph.D. degree in 1993 from the Vienna University of Technology. Since 2003, he has been a full professor of computer science in the Institute of Computer Science, University of Innsbruck, Austria. His main research interests include software architectures, programming paradigms, compiler technology, performance analysis, and prediction for parallel and distributed systems.