Fault Tolerance and Task Clustering in Scientific Workflows

Weiwei Chen [a,1], Rafael Ferreira de Silva [a], Ewa Deelman [a] and Thomas Fahringer [b]
[a] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[b] *University of Innsbruck, Institute for Computer Science, Innsbruck, Austria*

Abstract. Large scale scientific workflows can be composed of many fine computational granularity tasks. Task clustering has been proven to be an effective method to reduce execution overhead and increase the computational granularity of workflow tasks executing on distributed resources. However, a job composed of multiple tasks may have a greater risk of suffering from failures than a job composed of a single task. Our theoretic analysis and simulation results demonstrate that transient failures can have a significant impact on the runtime performance of scientific workflows that use existing clustering policies that ignore failures. We propose a general task failure modeling framework to address these performance issues. We show the necessity to consider the fault tolerance in the task clustering methods. We further propose three horizontal methods and two vertical methods to improve the runtime performance of executing workflows in dynamic environments. A trace based simulation is performed and it shows that our methods improve the workflow makespan significantly for five important applications.

Keywords. scientific workflows, fault tolerance, scheduling, locality, high availability

Introduction

Scientific workflows can be composed of many fine computational granularity tasks and the runtime of these tasks may be even shorter than the system overhead. System overhead is the period time during which miscellaneous work other than the users computation is performed. Task clustering [1,2,3,4,5,6] is a technique that merges several short tasks into a single job such that the job runtime is increased and the overall system overhead is decreased. However, existing clustering strategies ignore or underestimate the impact of the occurrence of failures on system behavior, despite the current and increasing importance of failures in large-scale distributed systems, such as Grids [7,8,9], Clouds [10,11,7] and dedicated clusters. Many researchers [12,13,14,15] have emphasized the importance of fault tolerance design and indicated that the failure rates in modern distributed systems are significant. Among all possible failures, we focus on transient failures because they are expected to be more prevalent than permanent failures [12]. For example, denser integration of semiconductor circuits and lower operating voltage levels

[1]Corresponding Author: Weiwei Chen, University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA; E-mail: weiweichen@acm.org.

may increase the likelihood of bit-flips when circuits are bombarded by cosmic rays and other particles [12].

In task clustering, a clustered job consists of multiple tasks. A task is marked as failed (task failure) if it is terminated by unexpected events during the computation of this task. If a task within a job fails, this job has a job failure, even though other tasks within this job do not necessarily fail. In a faulty environment, there are several options for reducing the influence of workflow failures. First, one can simply retry the entire job when its computation is not successful as in the Pegasus framework [8]. However, some of the tasks within the job may have completed successfully and it could be a waste of time and resources to retry all of the tasks. Second, the application process can be periodically checkpointed so that when a failure occurs, the amount of work to be retried is limited. However, the overheads of checkpointing can limit its benefits [12]. Third, tasks can be replicated to different nodes to avoid failures that are related to one specific worker node. However, inappropriate clustering (and replication) parameters may cause severe performance degradation if they create long-running clustered jobs. As we will show, a long-running job that consists of many tasks has a higher job failure rate even when the inter-arrival time of failures is long.

We propose three horizontal methods and two vertical methods to improve the existing clustering techniques (with job retry and task replication) in a faulty environment. The first horizontal solution dynamically adjusts the granularity or clustering size (number of tasks in a job) according to the estimated task failure rate. The second horizontal method retries the failed tasks within a job. And the third horizontal solution is a combination of the first two approaches. The vertical methods adjust the clustering size by dividing a failed job into two even retry jobs to reduce the job failures. We further analyze the runtime performance of combining horizontal methods and vertical methods in different ways. In this paper we view the sequence of failure events as a stochastic process and study the distribution of its inter-arrival times, i.e. the time between failures.

Our work is based on an assumption that the the distribution parameter of the inter-arrival time is a function of the type of task or the node the task is executed on. Samak [16] et al. have analyzed 1,329 real workflow executions across six distinct applications and concluded that the type and the host id of a job are among the most significant factors that impacted failures. Task type related failure is a type of failure that only occurs to some specific types of tasks. A node failure only occurs to some specific execution nodes. Compared to our prior work in [3], we add a parameter learning process to estimate the average task runtime, the average system overhead and the average inter-arrival time of failures. We adopt an approach of prior knowledge based Maximum Likelihood Estimation that has been recently used in machine learning. Knowledge about the parameters are modeled as a distribution with known super-parameters. The distribution of the prior and the posterior are in the same family if the likelihood distribution follows some specific distribution. For example, if the likelihood is a Weibull distribution and we model prior knowledge as an Inverse-Gamma distribution, then the posterior is also an Inverse-Gamma distribution. This simplifies the estimation of parameters and integrates the prior knowledge and runtime datasets gracefully.

The three horizontal methods were introduced and evaluated in [3] on two workflows. In this paper, we complement this previous paper by studying (*i*) the performance gain of using three horizontal methods over a baseline execution on a larger set of workflows; (*ii*) the performance impact of the variance of the average task runtime, system

overheads and the inter-arrival time of failures; (*iii*) the performance impact of proposed vertical clustering methods and their combination with horizontal methods.

The rest of this paper is organized as follows. Section 1 gives an overview of the related work. Section 2 presents our workflow and failure models. Section 3 details our fault tolerant clustering methods. Section **??** reports experiments and results, and the paper closes with a discussion and conclusions.

1. RELATED WORK

Failure analysis and modeling [13] presents system characteristics such as error and failure distribution and hazard rates. Schroeder et al. [14] has studied the statistics of the data, including the root cause of failures, the mean time between failures, and the mean time to repair. Sahoo et al. [15] analyzed the empirical and statistical properties of system errors and failures from a network of heterogeneous servers running a diverse workload. Oppenheimer et al. [17] analyzed the causes of failures from three large-scale Internet services and the effectiveness of various techniques for preventing and mitigating service failure. McConnel [18] analyzed the transient errors in computer systems and showed that transient errors follow a Weibull distribution. In [19,20] Weibull distribution is one of the best fit for the workflow traces they used. Based on these work, we measure the inter-arrival time of failures in a workflow and then provide methods to improve task clustering.

Overhead analysis [21,22] is a topic of great interest in the Grid community. Stratan et al. [23] evaluate in a real-world environment Grid workflow engines including DAGMan/Condor and Karajan/Globus. Their methodology focuses on five system characteristics: overhead, raw performance, stability, scalability, and reliability. They pointed out that head node consumption should not be negligible and the main bottleneck in a busy system is often the head node. Prodan et al. [22] offered a complete Grid workflow overhead classification and a systematic measurement of overheads. In Chen et al. [24], we extended [22] by providing a measurement of major overheads imposed by workflow management systems and execution environments and analyzed how existing optimization techniques improve runtime by reducing or overlapping overheads. The prevalent existence of system overheads is an important reason why task clustering provides significant performance improvement for workflow-based applications. In this work, we aim to further improve the performance of task clustering in a faulty environment.

Machine learning has been used to predict execution time [25,26,27] and system overheads [24], and develop probability distributions for transient failure characteristics. Duan et.al. [25] used Bayesian network to model and predict workflow task runtimes. The important attributes (such as the external load, arguments etc.) are dynamically selected by the Bayesian network and fed into a radial basis function neural network to make further predictions. In our paper, we reuse the knowledge (the family distribution) gained from prior work on failure analysis, overhead analysis and task runtime analysis. We then use prior knowledge based Maximum Likelihood Estimation to integrate both the knowledge and runtime feedbacks and adjust the estimation accordingly.

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, such as Grid and Cloud systems. Several works have addressed the control of task

granularity of bag of tasks. For instance, Muthuvelu et al. [28] proposed a clustering algorithm that groups bag of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [29] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [30] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [31] and Ang et al. [32] introduced bandwidth in the scheduling framework to enhance the performance of task scheduling. Longer tasks are assigned to resources with better bandwidth. Liu and Liao [33] proposed an adaptive fine-grained job scheduling algorithm to group fine-grained tasks according to processing capacity and bandwidth of the current available resources. Although these techniques significantly reduce the impact of scheduling and queuing time overhead, they did not consider fault tolerance.

Task granularity control has also been addressed in scientific workflows. For instance, Singh et al. [2] proposed a level- and label-based clustering. In level-based clustering, tasks at the same level can be 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. Although their work considers data dependencies between workflow levels, it is done manually by the users, which is prone to errors. Recently, Ferreira da Silva et al. [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. Their work significantly reduced scheduling and queuing time overheads, but did not consider fault tolerance.

2. Design and Models

2.1. Workflow Model

A workflow is modeled as a directed acyclic graph (DAG), where each node in the DAG often represents a workflow task (t), 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 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 workflow management systems such as Pegasus [8], Askalon [34], and Taverna [35]. In this paper, we assume there is only one execution site with multiple compute resources, such as virtual machines on the clouds.

Figure 1 shows a typical workflow execution environment that models a homogeneous computer cluster, for example, a dedicated cluster or a virtual cluster on Clouds. The submit host prepares a workflow for execution (clustering, mapping, etc.), and worker nodes, at an execution site, execute jobs individually. The main components are introduced below:

Workflow Mapper Generates an executable workflow based on an abstract workflow [8] provided by the user or workflow composition system. It also restructures the workflow to optimize performance and adds tasks for data management and provenance information generation. In this work, the workflow mapper is used to merge small tasks together

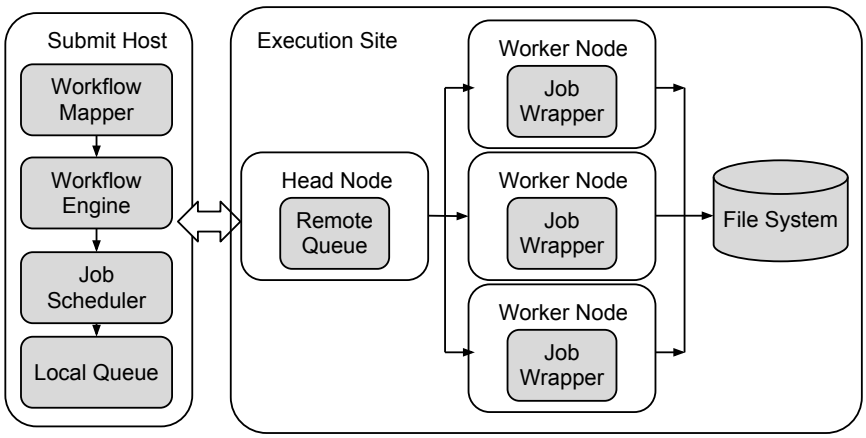

Figure 1. A workflow system model.

into a job such that system overheads are reduced (**task clustering**). A job is a single execution unit in the workflow execution systems and is composed of one or more tasks.

Workflow Engine Executes jobs defined by 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 elapsed time from when a job is released (all of its parents have completed successfully) to when it is submitted 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 elapsed time from when a task is submitted to the job scheduler to when it starts its execution in a worker node is denoted as the queue delay. It reflects both the efficiency of the job scheduler and the resource availability.

Job Wrapper Extracts tasks from clustered jobs and executes them at 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 [24]. Figure 2 shows how we augment a DAG to be an o-DAG with the capability to represent system overheads (s) such as workflow engine and queue delays. In addition, system overheads also include data transfer delay caused by staging-in and staging-out data. This classification of system overheads is based on our prior study on workflow analysis [24].

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

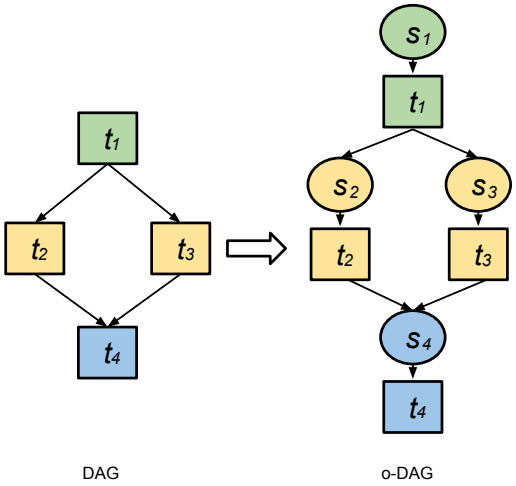

Figure 2. Extending DAG to o-DAG. 's' denotes a system overhead.

Figure 3 shows a simple example of how to perform HC, in which two tasks t_2 and t_3, without a data dependency between them, are merged into a clustered job j_1. A job j is a single execution unit composed by one or multiple task(s). Job wrappers are commonly used to execute clustered jobs, but they add an overhead denoted by 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 in one compute node is supported. In this work, we consider sequential executions only. Given a single resource, the overall runtime for the workflow in Figure 3 (left) is $runtime_l = \sum_{i=1}^{4}(s_i + t_i)$, and the overall runtime for the clustered workflow in Figure 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 $c_1 < s_3$, which is 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.

Figure 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 task clustering, the clustering size (k) is an important parameter to influence the performance. We define it as the number of tasks in a clustered job. For example, the k in Figure 3 is 2. The reason why task clustering can help improve the performance is that it can reduce the scheduling cycles that workflow tasks go through since the number of jobs has decreased. The result is a reduction in the scheduling overhead and possibly other overheads as well [24]. Additionally, in the ideal case without any failures, the clustering size is usually equal to the number of all the parallel tasks divided by the number of available resources. Such a nave setting assures that the number of jobs is equal to the number of resources and the workflow can utilize the resources as much as possible. However, when transient failures exist, we claim that the clustering size should be set based on the failure rates especially the task failure rate. Intuitively speaking, if

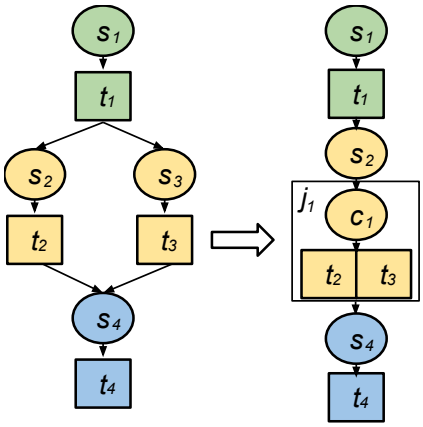

Figure 3. An example of horizontal clustering (color indicates the horizontal level of a task).

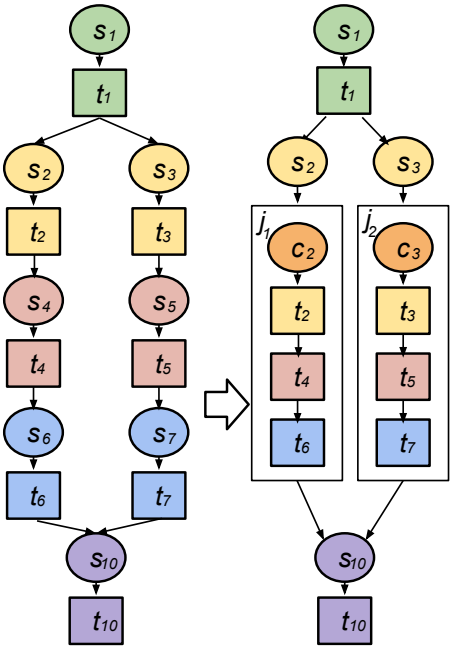

Figure 4. An example of vertical clustering.

the task failure rate is high, the clustered jobs may need to retry more times 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 average task runtime t, average system overhead s and the inter-arrival time of task failures γ.

2.2. Task Failure Model

In our prior work [24], we have verified that system overheads s fits Gamma or Weibull distribution better than the other two distributions (Exponential and Normal). Schroeder et. al. [14] have verified the inter-arrival time of task failures fits a Weibull distribution with a shape parameter of 0.78 or a gamma distribution better than lognormal and exponential distribution. We will reuse this shape parameter of 0.78 in our failure model. In [19,20] Weibull, Gamma and Lognormal distribution are among the best fit for the workflow traces they used. Without loss of generality, we choose Gamma distribution to model the task runtime ($\boldsymbol{t}$) and the system overhead ($\boldsymbol{s}$) and use Weibull distribution to model the inter-arrival times of failures ($\boldsymbol{\gamma}$). $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{\gamma}$ are all random variables of all tasks instead of one specific task.

In Bayesian probability theory, if the posterior distribution $p(\theta|D,a,b)$ are in the same family as the prior distribution $p(\theta|a,b)$, the prior and the posterior are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function. For example, 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 a Inverse-Gamma prior over the mean will ensure that the posterior distribution is also Inverse-Gamma. This simplifies the estimation of parameters since we can reuse the prior work from other researchers on the failure analysis and performance analysis.

After we observe data X, we compute the posterior distribution of θ:

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

$X = \{x_1, x_2, ..., x_n\}$ is the observed data of $\boldsymbol{\gamma}$ during the runtime. Similarly, we define $T = \{t_1, t_2, ..., t_n\}$ and $S = \{s_1, s_2, ..., s_n\}$ are the observed data of $\boldsymbol{t}$ and $\boldsymbol{s}$ respectively. $p(\theta|X,a,b)$ is the posterior that we aim to compute. $p(\theta|a,b)$ is the prior, which we have already known from previous work. $p(X|\theta)$ is the likelihood. We only need to know its shape parameter instead of its scale parameter, which reduces the work to estimate its scale parameter.

We model the inter-arrival time of failures ($\boldsymbol{\gamma}$) with a Weibull distribution that 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 Weibull distribution with a known shape parameter ϕ_γ is Inverse-Gamma distribution, which means if the prior 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:

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

The MLE (Maximum Likelihood Estimation) of the scale parameter θ_γ is:

$$MLE(\theta_\gamma) = \frac{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}{a_\gamma + n + 1} \tag{2}$$

The understanding of the MLE has two folds: initially we do not have any data and thus the MLE is $\frac{b_\gamma}{a_\gamma+1}$, which means it is determined by the prior knowledge; 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 it is close to the regularized average of the observed data.

We model the task runtime ($\boldsymbol{t}$) with a Gamma distribution 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 follows $\Gamma(a_t, b_t)$, while a_t is the shape parameter and b_t is 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 $\frac{b_t + \sum_{i=1}^{n} t_i}{a_t + n\phi_t - 1}$.

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 is $\Gamma(a_s, b_s)$, the MLE of θ_s is $\frac{b_s + \sum_{i=1}^{n} s_i}{a_s + n\phi_s - 1}$.

We have already assumed the task runtime, system overhead and inter-arrival time between failures are a function of task types. Since in scientific workflows, tasks at the different level (the deepest depth from the entry task to this task) are usually of different type, we model the runtime level by level. Given n independent tasks at the same level and the distribution of the task runtime, the system overhead, and the inter-arrival time of failures, we aim to reduce the cumulative runtime $\boldsymbol{M}$ of completing these tasks by adjusting the clustering size k (the number of tasks in a job). $\boldsymbol{M}$ is also a random variable and it includes the system overheads and the runtime of the clustered job and its subsequent retry jobs if the first try fails. The calculation of $\boldsymbol{M}$ does not consider the number of resources.

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 a system overhead. We assume the task runtime of each task is independent of each other, therefore the cumulative task runtime of k tasks is also a Gamma distribution. We also assume the system overhead is also independent of all the task runtimes and it has the same shape parameter ($\phi_{ts} = \phi_t = \phi_s$) with the task runtime. The sum of Gamma distributions with the same shape parameter is still a Gamma distribution. The job runtime irrespective of whether it succeeds of fails is:

$$\boldsymbol{d} \sim \Gamma(\phi_{ts}, k\theta_t + \theta_s) \tag{3}$$

$$MLE(\boldsymbol{d}) = (k\theta_t + \theta_s)(\phi_{ts} - 1) \tag{4}$$

Let the retry time of clustered jobs to be N. The process to run and retry a job is a Bernoulli trial with only two results: success or failure. Once a job fails, it will be retried

until it is eventually completed successfully since we assume the failures are transient. For a given job runtime d_i, by definition:

$$N_i = \frac{1}{1-F(d_i)} = \frac{1}{e^{-(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}}} = e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{5}$$

$F(x)$ is the CDF of $\boldsymbol{\gamma}$. The time to complete d_i successfully in a faulty environment is

$$M_i = d_i N_i = d_i e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{6}$$

Equation 6 has involved two distributions $\boldsymbol{d}$ and θ_γ. From Equation 1, we have:

$$\frac{1}{\theta_\gamma} \sim \Gamma(a_\gamma + n, \frac{1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}) \tag{7}$$

$$MLE(\frac{1}{\theta_\gamma}) = \frac{a_\gamma + n - 1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}} \tag{8}$$

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:

$$MLE(M_i) = MLE(d_i) e^{(MLE(d_i)MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{9}$$

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:

$$\boldsymbol{M} = \sum_{i=1}^{\frac{n}{k}} M_i \sim \Gamma$$

$$MLE(\boldsymbol{M}) = \frac{n}{rk} MLE(M_i) \tag{10}$$

$$= \frac{n}{rk} MLE(d_i) e^{(MLE(d_i)MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{11}$$

r is the number of resources. In our paper, we consider a compute cluster as a homogeneous cluster, which is usually true in dedicated clusters and Clouds.

Let k^* is the optimal clustering size that minimizes Equation 10. It is difficult to find a analytical solution of k^*. However, there are a few constraints that can simplify the estimation of k^*: (*i*) k can only be integers in practice; (*ii*) $MLE(\boldsymbol{M})$ is continuous and has one minimal. Methods such as Newton's method can be used to find the minimal $MLE(\boldsymbol{M})$ and the corresponding k. Figure 5 shows the three examples of $MLE(\boldsymbol{M})$ using 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). Other parameters are $n = 50$, $\theta_t = 5$ sec, and $\theta_s = 50$ sec. These parameters are close to the level of mProjectPP in the Montage workflow that we simulate in Section **??**. Figure 6 shows the relationship between the optimal clustering size (k^*) and θ_γ, which is a non-decreasing function. It is consistent with our expectation since 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 assures that we reduce system overheads without retrying many times.

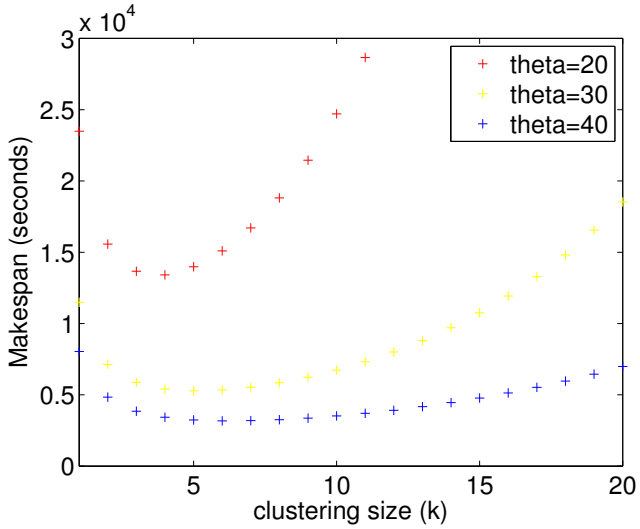

Figure 5. Makespan with different clustering size. ($n = 1000$, $r = 20$, $t = 5$ sec, $s = 50$ sec)

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; (*ii*), adjusting the clustering size according to the detected inter-arrival time can improve the runtime performance.

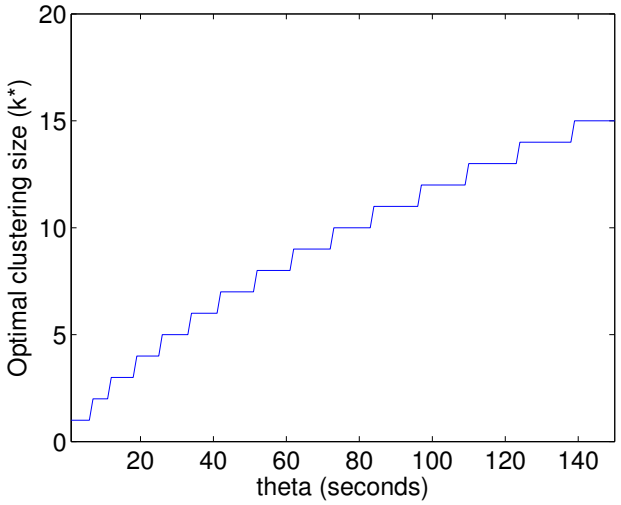

Figure 6. Optimal clustering size (k*) with different θ ($n = 1000$, $r = 20$, $t = 5$ sec, $s = 50$ sec)

3. Fault Tolerant Clustering

As indicated in Section 2.1, inappropriate clustering may increase the makespan of a workflow. To improve the fault tolerance from the point of view of clustering, we propose three horizontal methods: Dynamic Clustering (*DC*), Selective Reclustering (*SR*) and Dynamic Reclustering (*DR*) and two vertical clustering methods: Binary Clustering (*BC*) and Binary Reclustering (*BR*).

Dynamic Clustering (*DC*) sets the clustering size to be k^* during the runtime based on the MLE of task runtime, system overheads and the inter-arrival time of failures from records of jobs that have already been completed, either successfully or failed. Failure records contain the information about the number of failed tasks, the type of tasks, the resource id, and a timestamp. Figure 7 shows an example where the initial clustering size is 4 and thereby there are four tasks in a clustered job at the beginning. During execution, three out of these tasks (t_1, t_2, t_3) fail. Assuming our model suggests an optimal clustering size to be 2, then this job will be split into two clustered jobs while each has two tasks and the two clustered jobs are then submitted for retry. DC is not aware that there are only three failed tasks. This approach requires only a reclustering process that divides the original clustered job (failed) into multiple smaller clustered jobs based on the suggested k^*. It does not require to record which tasks exactly fail during the runtime, which makes it applicable to existing workflow management systems such as Pegasus [8].

Selective Reclustering (*SR*). In DC, the clustering engine knows the MLE of inter-arrival time of task failures but the information about which tasks have failed is not available. In practice, it may be hard to identify the failed tasks, but if it is supported, we can further improve the performance with Selective Reclustering that selects the failed tasks in a clustered job and clusters them into a new clustered job, or treats them individually. SR is different to the nave job retry in that the latter method retries all tasks of a failed job even though some of the tasks have succeeded. SR requires collecting the task ids in failure records.

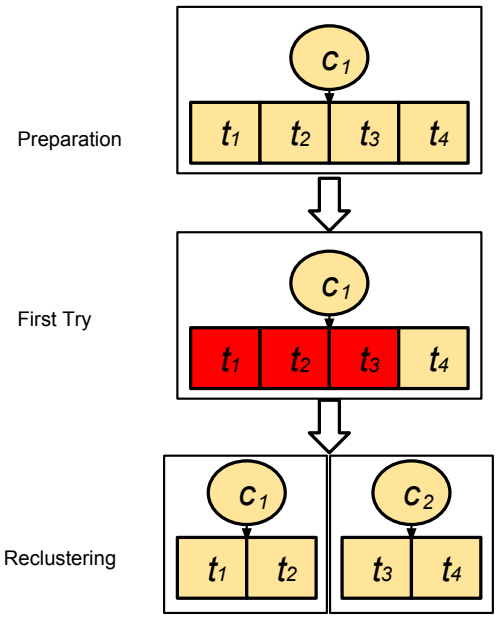

Figure 7. Dynamic Clustering (red boxes are failed jobs)

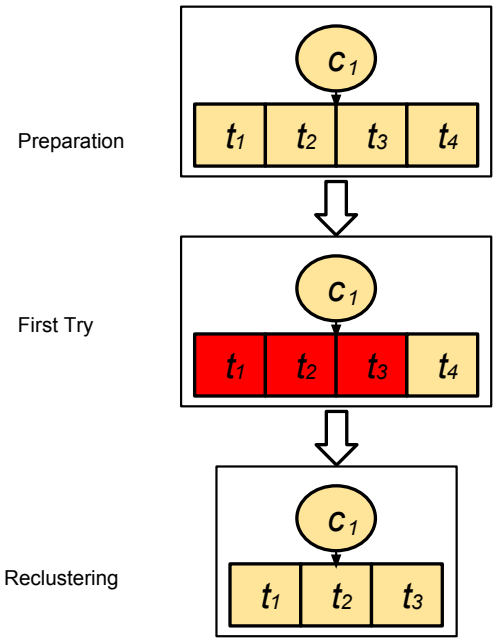

Figure 8. Selective Reclustering (red boxes are failed jobs)

Figure 8 shows an example of SR. At the first try, there are four tasks and three of

them have failed. One task succeeds and exits. Only the three failed tasks are merged again into a new clustered job and the job is retried. This approach does not intend to adjust the clustering size, although the clustering size will be smaller and smaller naturally after each retry since there are less and less tasks in a clustered job. In this case, the clustering size has decreased from 4 to 3. However, the optimal clustering size may not be 3, which limits its performance if the γ is small and k should be decreased as much as possible.

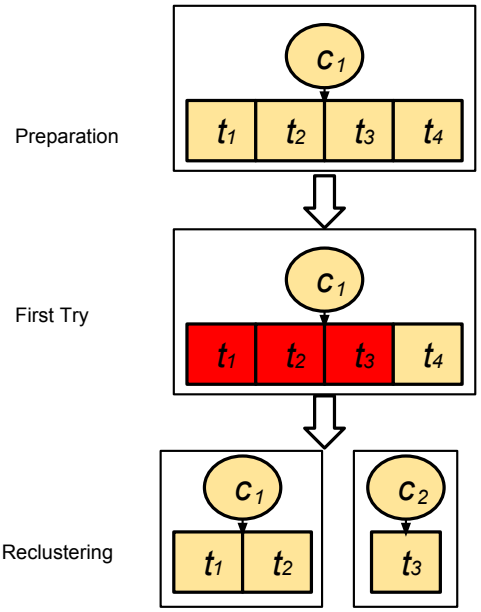

Figure 9. Dynamic Reclustering (red boxes are failed jobs)

Dynamic Reclustering (*DR*). Selective Reclustering does not analyze the clustering size rather, it uses a natural way to reduce the clustering size if the failure rate is too high. However, it requires a special ability to select the failed tasks, while DC does not. We then propose the third method, Dynamic Reclustering, which is a combination of SR and DC to see whether using both strategies can improve workflow performance. In DR, only failed tasks are merged into new clustered jobs and the clustering size is also adjusted according to the detected task failure rates. DR also requires the ability to detect which tasks have failed in a job.

Figure 9 shows the steps of DR. At the first try, three tasks within a clustered job have failed. Therefore we have only three tasks to retry and further we need to decrease the clustering size (in this case it is 2) accordingly.

Binary Clustering (*BC*) is used in vertical clustering to adjust the clustering size. If there is a failure detected, we decrease the k by half and reclusters them. In Figure 10, if there is no assumption of failures initially, we put all the tasks (t_1,t_2,t_3,t_4) from the same pipeline into a clustered job. t_3 fails at the first try assuming it is a failure-prone task (its γ is short). BC then divides the failed job into two even jobs and submits them again. In the second try, j_2 unfortunately fails again while j_1 succeeds and exits. Since

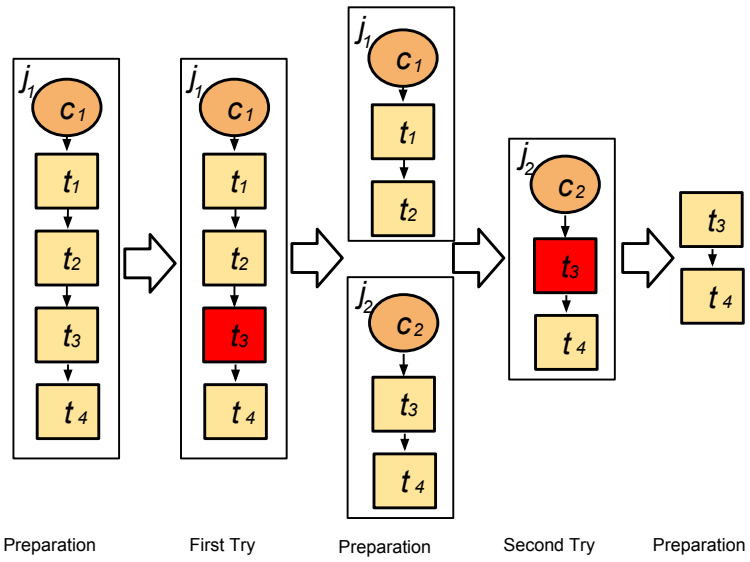

Figure 10. Binary Clustering (red boxes are failed jobs)

the clustering size is already 2, BC performs no vertical clustering anymore and would continue retrying t_3 and t_4 (but still following their data dependency) until they succeed.

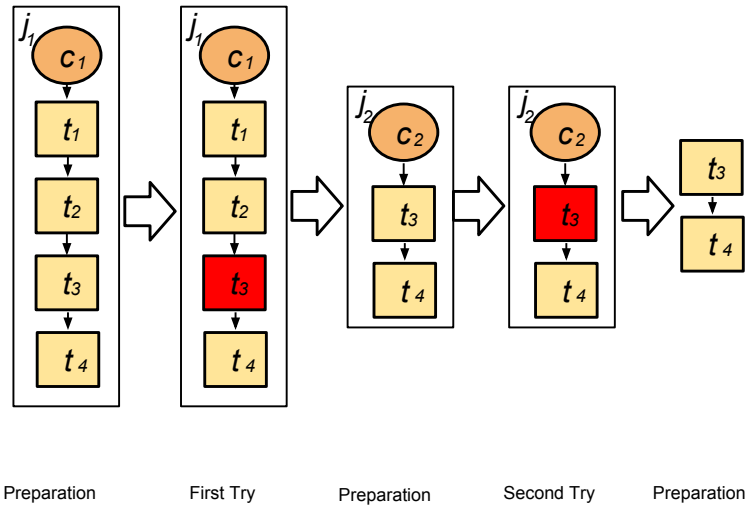

Figure 11. Binary Clustering (red boxes are failed jobs)

Binary Reclustering (*BR*). Similar to Selective Reclustering, Binary Reclustering only retries failed tasks. It also reduces the clustering size by half in each retry, which is similar to Binary Clustering. In Figure 11, after the first try, BR only retries t_3 and t_4 and put them into a new job. Other process are the same as Binary Clustering.

References

[1] W. Chen, E. Deelman, R. Sakellariou, Imbalance optimization in scientific workflows, in: Proceedings of the 27th international ACM conference on International conference on supercomputing, ICS '13, 2013, pp. 461–462.

[2] 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.

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

[4] K. Maheshwari, A. Espinosa, D. S. Katz, M. Wilde, Z. Zhang, I. Foster, S. Callaghan, P. Maechling, Job and data clustering for aggregate use of multiple production cyberinfrastructures, in: Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date, DIDC '12, ACM, New York, NY, USA, 2012, pp. 3–12.

[5] R. Ferreira da Silva, T. Glatard, F. Desprez, On-line, non-clairvoyant optimization of workflow activity granularity on grids, in: F. Wolf, B. Mohr, D. Mey (Eds.), Euro-Par 2013 Parallel Processing, Vol. 8097 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 255–266.

[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, 2012.

[7] J. Bresnahan, T. Freeman, D. LaBissoniere, K. Keahey, Managing appliance launches in infrastructure clouds, in: Teragrid Conference, 2011.

[8] 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.

[9] R. Duan, R. Prodan, T. Fahringer, Run-time optimisation of grid workflow applications, in: 7th IEEE/ACM International Conference on Grid Computing, 2006, pp. 33–40.

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

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

[12] 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.

[13] D. Tang, R. K. Iyer, S. S. Subramani, Failure analysis and modeling of a vaxcluster system, in: Proceedings of the International Symposium on Fault-tolerant computing, 1990.

[14] B. Schroeder, G. A. Gibson, A large-scale study of failures in high-performance computing systems, in: Proceedings of the International Conference on Dependable Systems and Networks, 2006.

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

[16] 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.

[17] 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.

[18] 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.

[19] X.-H. Sun, M. Wu, Grid harvest service: a system for long-term, application-level task scheduling, in: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 2003, p. 8.

[20] A. Iosup, O. Sonmez, S. Anoep, D. Epema, The performance of bags-of-tasks in large-scale distributed systems, in: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC '08, ACM, New York, NY, USA, 2008, pp. 97–108.

[21] P.-O. Ostberg, E. Elmroth, Mediation of service overhead in service-oriented grid architectures, in: Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on, 2011, pp. 9–18.

[22] R. Prodan, T. Fabringer, Overhead analysis of scientific workflows in grid environments, in: IEEE Transactions in Parallel and Distributed System, Vol. 19, 2008.

[23] C. Stratan, A. Iosup, D. H. Epema, A performance study of grid workflow engines, in: Grid Computing, 2008 9th IEEE/ACM International Conference on, IEEE, 2008, pp. 25–32.

[24] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: The 6th Workshop on Workflows in Support of Large-Scale Science, 2011.

[25] R. Duan, F. Nadeem, J. Wang, Y. Zhang, R. Prodan, T. Fahringer, A hybrid intelligent method for performance modeling and prediction of workflow activities in grids, in: Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on, 2009, pp. 339–347.

[26] J. Cao, S. Jarvis, D. Spooner, J. Turner, D. Kerbyson, G. Nudd, Performance prediction technology for agent-based resource management in grid environments, in: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, 2002, p. 14.

[27] H. Li, D. Groep, L. Wolters, Efficient response time predictions by exploiting application and resource state similarities, in: Grid Computing, 2005. The 6th IEEE/ACM International Workshop on, 2005, p. 8.

[28] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, R. Buyya, A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids, in: Proceedings of the 2005 Australasian workshop on Grid computing and e-research, 2005.

[29] N. Muthuvelu, I. Chai, C. Eswaran, An adaptive and parameterized job grouping algorithm for scheduling grid jobs, in: Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on, Vol. 2, 2008, pp. 975 –980.

[30] 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 Lecture Notes in Computer Science, 2010, pp. 266–277.

[31] 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.

[32] T. L. L. P. T.F. Ang, W.K. Ng, C. Liew, A bandwidth-aware job grouping-based scheduling on grid environment, Information Technology Journal (8) (2009) 372–377.

[33] Q. Liu, Y. Liao, Grouping-based fine-grained job scheduling in grid computing, in: First International Workshop on Education Technology and Computer Science, 2009.

[34] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, H. Truong, ASKALON: a tool set for cluster and grid computing, Concurrency and Computation: Practice & Experience 17 (2-4) (2005) 143–169.

[35] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.