TOWARD PRACTICAL MULTI-WORKFLOW SCHEDULING IN CLUSTER AND GRID ENVIRONMENTS

by

ZHIFENG YU

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2009

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor **Date**

_____

_____

_____

_____

ACKNOWLEDGEMENTS

Many people helped and supported me throughout this long journey. I am especially thankful to my advisor Dr. Weisong Shi for his paramount guidance and encouragement without that this dissertation is not possible. His help is always accessible and invaluable from directions of research to the discussions on details.

I am very thankful to Dr. Vaclav Rajlich for his invaluable guidance in getting my graduate study started on a solid foundation. His understanding and encouragement are sincerely appreciated. I also like to thank my committee members, Dr. Monica Brockmeyer and Dr. Song Jiang for their time and guidance in this process.

This work also benefits from the helps from Zhengqiang Liang, Kewei Sha, Chenjia Wang, Hanping Lufei, Safwan Al-Omari, Yong Xi, and other colleagues at Mobile and Internet SysTems Laboratory (MIST Lab). Their enthusiasm and hard work inspire me to make this endeavor finally successful. I am also grateful to Sam Corona, my colleague at T-Systems, for his review and comments.

Finally, I would like to thank my family, my parents and parents in law for the supports throughout so many years. I will be eternally indebted to my wife, Xian, for her love, encouragement, support and sacrifice. I am equally indebted to my daughter, Daisey, who brings so much joy and excitement to my life. And it is too young for her to understand now how much I actually learned from her to be patient and determined, which makes this dissertation a reality today.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Emergence of Scientific Workflow Applications

A workflow application is a set of tasks or jobs which are coordinated by control and data dependencies to accomplish complex composite work. In 2006, the National Science Foundation brought together domain, computer and social scientists to discuss the challenges of scientific workflows, which recognized the critical role of scientific workflows in cyberinfrastructure and recommended that "workflows should become first-class entities in cyberinfrastructure architecture" [47]. This dissertation primarily focuses on scientific workflow applications in cluster and Grid context.

Workflows have recently emerged as a paradigm for representing and managing complex distributed scientific computing, accelerating the pace of scientific progress [28,47]. Thanks to the growing popularity of cluster and Grid platforms (TeraGrid [4], Planet Lab [90], EuroGrid [34], Open Science Grid [5] etc.) which now play an increasingly important role for supporting computation execution in a widely distributed environment, workflow applications have become even more popular. Much scientific research work which requires intensive computation or data analysis already leverage the complex workflow applications running on Grid platform, such as GriPhyN [50] for experimental physics, WIEN2K [114] for quantum chemistry, Montage [3] for astronomy, EMAN [2] for electron micrograph analysis, LEAD [64] for meteorological data analysis and weather forecasting etc.

On the other hand, the growing popularity of workflow applications keeps pushing technology limit of workflow management systems, which define, manage and execute complex workflows on heterogeneous distributed computing environments. Recent years have

witnessed design and development of many workflow management systems, such as Pegasus [89], GrADS [49], DAGMan/Condor [1], Taverna [83], GridFlow [18], ASKLON [113] etc. Some of these systems are designed for specific scientific domain problems, while others are developed for generic usage.

One of the greatest challenges is scaling of a workflow management system [47], i.e., when individual workflow becomes more complex, when number of participants, workflows and resources increase, how to schedule and manage workflow execution on shared resources to achieve high performance and scalability with infrastructure constraints. Even though tremendous efforts are spent on research and practice, it is yet far from resolution. At the core of this challenge is workflow scheduling, which is a classic problem but elevated to a much higher level of complexity in context of Grids.

1.2 Workflow Scheduling Problems

Workflow scheduling is recognized as an NP-Complete problem even with minimum multiple processors [46]. The inherent characteristics of a Grid, such as large scale, heterogeneity, loose control of resource, resource volatility in terms of availability and capability, contribute to greater complexity.

In cluster and Grid environments, resource set is typically heterogeneous even within the same class and type of resources [39]. Each resource may have different provisioning capability and failure rate. In addition, the resources from different virtual organizations constitute a loosely organized grid platform, without centralized ownership and control.

The problem of scheduling workflow in cluster and Grid environments is to schedule each individual task of a workflow application to a suitable resource collection, complete the entire worklfow with minimal *makespan* and *turnaround* time, while preserving task dependencies and meeting data transfer requirements, in a heterogeneous and dynamic environment. In reality, many Grid systems are built by clustering multiple clusters. The scheduling strategy proposed in this dissertation does not only apply to Grids but also clusters. Henceforth we will refer to the target environment, i.e., cluster and Grid environment, simply as "Grid".

The objective of this dissertation is to design an efficient and practical scheduling scheme to optimize workflow application performance, which answers the following critical questions and directly addresses issues challenging workflow management scalability identified in [28]:

1. How to schedule a complex workflow in a dynamic heterogenous Grid?

2. How to schedule multiple workflows with dynamic workload in a multiple user Grid?

3. How to schedule workflows in an error prone computing environment?

4. Finally, how to schedule workflows in a vast infrastructure which is built upon existing clusters and grids.

1.3 Motivations

When it comes to workflow management design, performance is an ultimate success criteria. As a key design component of the system, a good scheduler ensures shared resources are efficiently utilized to execute workflows and achieve optimal performance, measured by *makespan*, *turnaround* time and *resource utilization efficiency* from perspectives of both user and system.

Grid systems introduce great challenges to general resource management. The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [41]. At the heart of the Grid is the ability to discover, allocate, and negotiate the use of network-accessible capabilities, which is referred to as resource management [23]. What makes resource management in Grid systems distinct from traditional computation platform is that resources in typical grid systems are heterogeneous, dynamic, volatile and controlled in a decentralized way. Unfortunately, as a result, most workflow scheduling heuristics developed for traditional platforms are not applicable to Grid systems.

Moreover, a workflow application adds additional complexity to grid resource management challenge because of the inter-task dependencies and data staging requirements. It is

also observed that the workflow complexity keeps increasing, for example, number of tasks increases to new magnitude, data movement crosses domains and even workflow itself is dynamically built on the fly. In a more complex case, many participants might help define the workflow, contributing relevant data, managing its execution and interpreting results [47]. The last decade witnessed tremendous research effort on how to define, design, optimize workflow and data transition, which resulted in a number of Grid workflow management systems. However, grid workflow scheduling remains a core challenge, yet far from being well addressed.

However, because of the NP-Complete difficulty, most existing static scheduling strategies simplify the scheduling problem to various extents. The most common assumptions are listed as below:

1. Number of workflows. Most studies assume that there is only one workflow application executing in a grid environment, referred to as single workflow scheduling problem. But in the real world, a typical grid system may serve multiple users and multiple applications concurrently. Particularly in large communities, many users might submit many workflows at once [47] and the workload is very dynamic in terms of number of users, number of workflows and mixture of workload.

2. No resource competition. This oversimplification is derived from the previous assumption of single workflow scheduling. As there is only one workflow application, it does not have to compete for resources with other applications and it is assumed that resources are always 100% dedicated and available once the job is mapped to the particular resource.

3. Dynamic resource. It is very typical that resources come and go in a grid system. Existing static scheduling strategies assume that resource set is given and fixed over time. The assumption is not always valid even with the reservation capability in

place. Moreover, the static scheduling approach neither considers new resources after the plan is made nor resource decommission during the course of execution.

4. Accurate performance estimation. Estimating communication and computation costs of a workflow is the key success factor for any static approach but practically difficult. The deviation in run time is detrimental to estimation based static scheduling. Resource mapping based on inaccurate estimation performs even worse than a simple random algorithm.

5. No resource failure during execution. Software and hardware failure is very common in a grid system. An unpredicted failure incident may fail a static plan in some cases. However, almost none of the static approaches considers the failure impact on the static plan. Therefore, the workflow performance can easily turn out to be very poor as unexpected resource failures force workflow execution derail from the original schedule.

6. Centralized resource control. Another important assumption is the centralized control over resources. This is a valid assumption for an isolated cluster environemnt. However, nowadays the magnitude of Grid is mostly achieved by clustering existing clusters. It does not warrant a centralized scheduler which can directly schedule a job to a specific resource collection. Instead, jobs are dispatched to a cluster which in turn schedules the job according to local scheduling policy independently and local scheduling policies may vary from one cluster to another.

Recognizing that the unrealistic assumptions made by static strategy lead to its impracticality, some workflow management systems employ just-in-time scheduling, i.e. dynamic approach. This helps to address some invalid assumptions made by static approaches, however it performs inferiorly to some static scheduling approaches even when job performance estimation is not very accurate [113]. Without prior knowledge of workflow structure and

performance estimation, dynamic scheduling makes local decisions and therefore is myopic. The simulation work [14] further suggests that static approaches still perform better than dynamic ones for data intensive workflow applications even with inaccurate information about future jobs.

We argue that static scheduling strategies and dynamic ones are not mutually exclusive. It is thus motivated to adapt the static strategies to a dynamic grid environment by designing a planner guided dynamic scheduler which tackles dynamics of both resource and workload, and targets at improving efficiency, practicability of workflow scheduling and efficiency of resource utilization.

1.4 Contributions

The primary contribution of this dissertation is a proposed collaborative workflow scheduling strategy, which exploits advantages of both static and dynamic strategies to achieve the goal of workflow performance from perspectives of both system and user. The novelties of proposed strategy are:

1. Design a collaborative workflow scheduling system which realizes the benefit of static approaches by collaborating workflow *Planner* with *Executor*. With this approach, the *Planner* first prioritize all tasks and even proposes a resource mapping. During the execution, the *Executor* will notify the *Planner* of any run time event which interests the *Planner*, for example, resource unavailability, discovery of new resource or estimation refining etc. In turn, the *Planner* responds to the event by means of evaluating the event and rescheduling the remaining jobs in the workflow if necessary. Planning is now an iterative (event-driven) activity instead of one time task.

2. The system described above employs a planner-guided priority based dynamic scheduling algorithm. The *Planner* assigns each individual task within a wokflow a rank value based on any list based static heuristic. The *Executor* maintains and manages a global *Job Pool* of ready to execute jobs from multiple workflows. Taking the local rank value

of each task possibly from different users, the *Executor* re-assigns each task a global priority and schedules them based on priority. It not only inherits the advantages of static scheduling strategy, but also leverages dynamic scheduling to address issues of practicability, dynamic resource and workload.

3. The system design also supports failure aware workflow scheduling, which allows the system schedule efficiently in an error prone computation environment. The research studies how error prediction accuracy affects scheduling efficiency, which in turn determines the workflow performance. It argues that schedulers have different requirements on failure prediction. The recognition of the difference helps to define a reasonable accuracy requirement for resource failure prediction. More importantly, the simulation results prove that workflow application performance can be improved with an achievable failure prediction accuracy.

4. Scheduling workflows in multicluster environments. Nowadays, as the infrastructure matures rapidly, multicluster environments emerge to meet increasing computation demands. These clusters are independent of each other and have their own local job scheduler, each with potentially a different policy. While such shared infrastructure provides explosive computation capability, it is a great challenge to schedule workflows in such environment to achieve better performance while observing the local job scheduling policies. The system design proposed in this dissertation leverages the advancement of queue waiting time prediction techniques and helps a global scheduler dispatch jobs to various clusters with data movement, queue waiting time and cluster capability in consideration. The extensive simulation result proves that the approach improves not only the workflow performance but also resource utilization efficiency.

In addition to the above novelties, this dissertation implements a workflow scheduler prototype with Condor/Condor-G [22, 43], which combines the inter-domain resource management protocols of the Globus Toolkit and the intra-domain resource and job management

methods of Condor to allow the user to harness multi-domain resources as if they all belong to one personal domain. A prototype of the proposed workflow scheduler is developed in Java and integrated with Condor-G with provided Condor Web Service interfaces.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 provides the background of problem domain of workflow scheduling in Grids and reviews the related work in this area. Chapter 3 describes the system design of a proposed collaborative workflow scheduling system. The subsequent chapters elaborate various aspects of the proposed solution, including Chapter 4 which describes an adaptive rescheduling scheme, Chapter 5 which provides details on how to schedule multiple workflows, Chapter 6 which presents how failure aware scheduling is supported and Chapter 7 which demonstrates how to schedule workflows in shared infrastructure such as a cluster of clusters. Chapter 8 presents the prototype design and development based on Condor-G. Finally, Chapter 9 summarizes the work in this dissertation and discusses future work in this area.

CHAPTER 2
BACKGROUND OF WORKFLOW SCHEDULING

In this chapter, Section 2.1 first provides an overview of workflow scheduling modeling. Section 2.2 discusses the target computing environment in study. Workflow scheduling performance criteria are covered in Section 2.3. And an overview of workflow scheduling research is presented in Section 2.4.

2.1 Workflow Scheduling Modeling

A workflow scheduling system model consists of a workfow application, a target computing environment, and performance criteria for scheduling [111].

A typical scientific workflow application can be represented as a Direct Acyclic Graph(DAG), a directed graph with no directed cycles. In a DAG, a node is an individual task and an edge represents the inter-job dependency. With the dependency, which can be either control dependency or data dependency, a child task can not be executed before all its parent tasks finish successfully and its required data input is in place. Nodes and edges are weighed for computation cost and communication cost respectively. A DAG is referred to as Task Graph in other related research.

It is worth noting that not all workflows can be modeled as a DAG but majority of scientific workflows can be so. This dissertation focuses on DAG-based scientific workflows only. In the rest of this dissertation, the terms DAG and workflow are used interchangeably.

In order to be consistent with other related work, the model defined for workflow scheduling in a heterogeneous environment by Topcuouglu *et al.* [111] is used in this dissertation with minor revision.

A DAG can be modeled as a tuple $G = (V, E)$ where $V = \{n_j, j = 1..v\}$ is the set of task(job) nodes (Task, job and node are interchangeable terms in this dissertation), E is the set of communication edges between tasks. Each edge $(i, j) \in E$ represents the precedence constraint that task n_j can not start without successful completion of task n_i. $Data$ is a $v \times v$ matrix of communication data, where $data_{i,j}$ is the amount of data required to be transmitted from task n_i to task n_j.

The target computing platform is defined as R, a set of r heterogeneous processors, which are connected in a fully connected network topology. The data transfer rate, i.e. bandwidth, between processors is defined in a matrix B of size $r \times r$. Section 2.2 provides more details about the target computing platform.

It is assumed in this model that computation can overlap with communication. W is a $v \times q$ matrix of computation cost matrix in which each $w_{i,j}$ represents the execution time to complete task n_i on processor r_j. The average execution cost of a task n_i is defined as

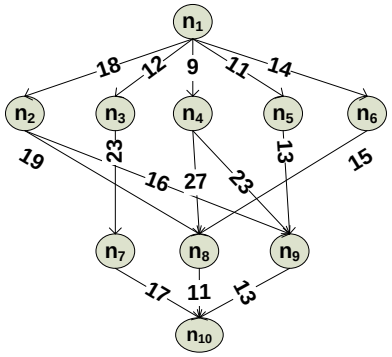
$$\overline{w_i} = \sum_{j=1}^{q} \frac{w_{i,j}}{q} \tag{2.1}$$

A r-dimensional vector L stores the communication startup latency for each processor. The communication cost, latency for data transition of the edge (i, k), which is for transferring data from from task n_i (executed on processor r_m) to task n_k (executed on processor r_n) is defined as:

$$c_{i,k} = L_m + \frac{data_{i,k}}{B_{m,n}} \tag{2.2}$$

Typically modeling further reasonably simplifies the communication cost by introducing an average communication cost of an edge (i, k) as defined by:

$$\overline{c_{i,k}} = \overline{L} + \frac{data_{i,k}}{\overline{B}}, \tag{2.3}$$



Computation Cost

Job	Resource			
	r_1	r_2	r_3	r_4
n_1	14	16	9	*14*
n_2	13	19	18	*17*
n_3	11	13	19	*14*
n_4	13	8	17	*15*
n_5	12	13	10	*14*
n_6	13	16	9	*16*
n_7	7	15	11	*15*
n_8	5	11	14	*20*
n_9	18	12	20	*13*
n_{10}	21	7	16	*15*

Figure 2.1: A sample DAG.

where $\overline{B}$ is the average transfer rate among the processors in the domain and $\overline{L}$ is the average communication startup time.

A DAG example is shown Figure 2.1, which models a DAG and a target grid system with four processors and corresponding communication and computation costs. In the figure, the weight of each edge represents its average communication cost.

2.2 Target Computing Platform

The complex workflows typically execute in a high performance cluster or grid environment. As defined in [12], a cluster is a type of parallel or distributed processing system which consists of a collection of interconnected stand alone computers working together as a single integrated computing resource. A computer node can be a single or multiprocessor system (PCs workstations or SMPs) with memory I/O facilities and an operating system. A typical cluster generally refers to two or more computers nodes connected together via LAN-based network, as Figure 2.2 shows. Figure 2.3 illustrates the typical cluster architecture in TeraGrid infrastructure [4]. It is very important that a cluster is envisioned as one single resource from user perspective, facilitated by cluster middleware.

As defined by Foster [37], a grid coordinates resources that are not subject to centralized control, uses standard, open, general-purpose protocols and interfaces and delivers nontrivial

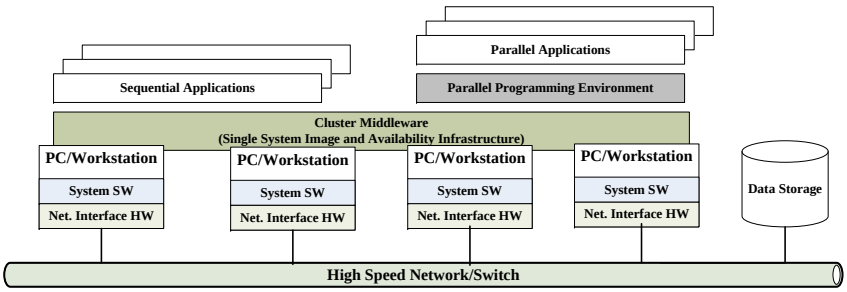


Figure 2.2: Conceptual cluster architecture [12].

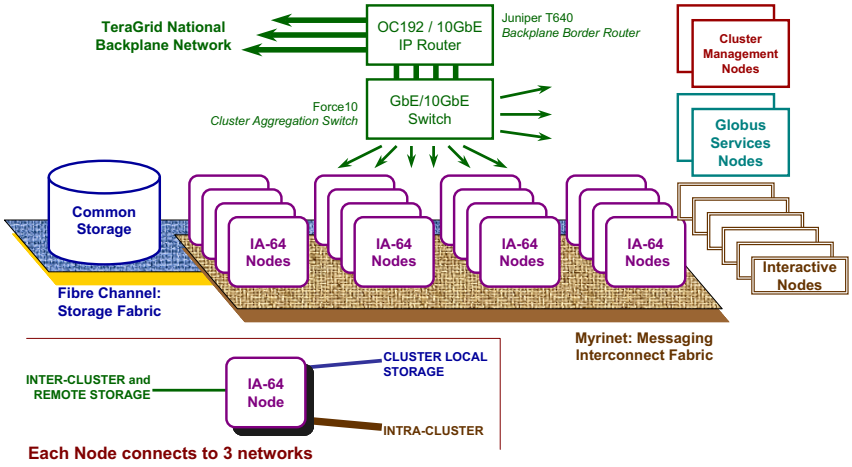


Figure 2.3: TeraGrid cluster architecture [20].

qualities of service. It is composed of a cluster of networked, loosely-coupled computers of heterogeneity. In real world, many Grids are in form of a cluster of clusters, where clusters are connected with high bandwidth backbone and each cluster has its own independent workload management system. TeraGrid [4] is a typical example, as shown in Figure 2.4, which connects clusters of Caltech, ANL, SDSC, NCSA and PSC etc..

In our abstract model, the target computing environment is a pool of heterogenous resources. The resource can be generically referred to as Target Processor Element(TPE) [54], which represents a single computing unit. A TPE can be a stand-alone PC, a processor or a cluster. As workflow typically involves data stage-in and stage-out, it is safe to assume that

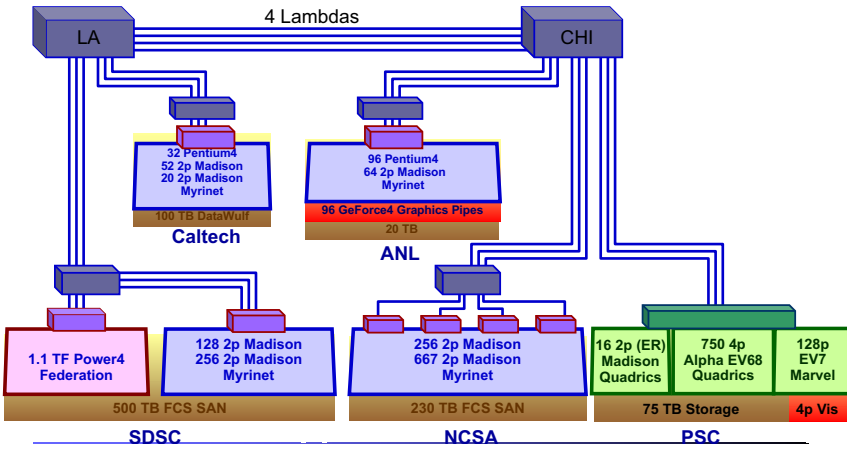


Figure 2.4: TeraGrid architecture [20].

only resources within a domain will share file storage server. Otherwise, data movement will occur over the network.

In the target computing platform, resources are shared in a fashion of space sharing, as most of High Performance Computing(HPC) systems implement today. A space sharing enforced scheduling policy allocates each job a dedicated set of resources for the duration of its execution. Once scheduled, the job does not share its allocated resource with other jobs and can not be preempted during its execution.

As this dissertation focuses on workflow scheduling, it is assumed that the Grid fabric service, resource and connectivity protocol implementation and collective services defined in Grid architecture [40] are in place and readily to use. It is also assumed that jobs can be executed remotely on any resource in the pool when workflow is submitted by a user. For instance, when a user submits a job to Condor, the job can be executed on any remote machine within the pool of machines available to Condor without modifying the source code [22].

2.3 Performance Criteria

Performance criteria is another component in workflow scheduling system modeling. It is used to evaluate the effectiveness of the scheduling strategy. As some criteria may conflict with each other, any system design can not accommodate all criteria at same time and reasonable trade offs have to be made. In this dissertation, the following criteria are chosen to justify the proposed scheduling strategy:

1. *Makespan.* Also referred to as schedule length, it is the time difference between workflow start and its completion. Most DAG algorithms use *makespan* to evaluate their effectiveness from the perspective of workflow applications. The smaller the *makespan*, the better performance achieved.

2. *Turnaround.* This is the time difference between workflow submission and its completion. Different than *makespan*, *turnaround* time includes the time workflow application spends waiting to get started. It is used to measure the performance and service salification from user perspective. This measurement is rarely used in traditional DAG scheduling as it does not consider resource competition. In a real world Grid system, the workflow application submitted by a user may not get started instantly, depending on resource availability and scheduling policy.

3. *Resource effective utilization.* From a system perspective, resource management looks for effective utilization of all resources. Similar to [97], it is defined as:

$$Resource\ Effective\ Utilization = \frac{\sum ProcessorUsed_i \times Runtime_i}{ProcessorsAvailable \times Makespan} \tag{2.4}$$

   The goal of higher *resource effective utilization* is correlated to the objectives of lower *makespan* and *turnaround* time, because the latter two can not be achieved without efficient usage of all available resources.

4. *Resource allocation fairness.* When multiple workflow applications execute concurrently in a shared resource environment, the scheduling policy should treat each of them fairly and prevent resource starvation from occurring. The *fairness* can be measured by the ratio of the total amount of time the ready to execute jobs are waiting for resource and the *turnaround* time. A fair scheduling policy ensures the ratio for each workflow application is relatively close.

2.4 Overview of Workflow Scheduling in Grids

2.4.1 Existing Scheduling Algorithms

Generally speaking, a workflow scheduling strategy can be categorized as either static or dynamic. Static scheduling maps resources to each individual task before workflow execution, it favors the entire workflow performance but relies on the knowledge of workflow applications and execution environments. Conversely, dynamic scheduling makes a resource mapping decision only when a task is ready to execute without requiring any prior application and environment knowledge.

In dynamic scheduling, the task dependency is managed by the job submission manager which enforces task dependencies. The DAG task in the queue meets the dependency requirement and is not differentiated from other independent jobs. For example, in the Condor system, DAGMan [1] checks the inter-task dependency and submits only the task ready to execute into the queue managed by Condor [21] which schedules the jobs in the queue in first come first service (FCFS) fashion.

With regard to static heuristics of DAG scheduling in a heterogeneous environment, there are two groups that are more popular than others: List scheduling heuristics and clustering heuristics. A list scheduling heuristic maintains a list of all tasks of a given DAG according to their priorities. It firstly prioritizes or ranks all tasks, then selects best resource (by defined cost objective function) for the ready task with highest priority. The well known heuristics in this group [32, 59, 61, 103, 104, 111, 116] are Heterogeneous

Earliest Finish Time (HEFT) [111], Dynamic Critical Path (DCP) [61], and Dynamic Level Scheduling (DLS) [103] etc.

The clustering algorithm involves a two step scheduling approach. The first step is the clustering phase: tasks are grouped into clusters on virtual processors for certain criteria, for example, in order to suppress unnecessary communications. The second one is the mapping phase: all member task of same cluster are assigned to same processor. Dominant Sequence Clustering (DSC) [120] and Clustering and Scheduling System (CASS) [69] etc. belong to this group.

Extensive comparative studies [14,15,17] are performed and they show that static strategy can potentially perform near optimal, and this is also proven true with some real world workflow applications [113]. The simulation work [14] further suggests that static approaches still perform better than dynamic ones for data intensive workflow applications even with inaccurate information about future jobs.

However, it is very difficult to estimate execution performance accurately, since the execution environment may change a lot after resource mapping. The challenge with static strategy is discussed in research [25]. Recent work [35,76,108] shows that scheduling through resource reservation and performance modeling can help to ensure the resource availability during execution and theoretically makes the grid more predictable, but their approaches do not resolve all the problems. Others try to make static approach more adaptable to change occurring in execution phase. Rescheduling is implemented in the GrADS [13], where it is normally activated by contract violation. However, the efforts are all conducted for iterative applications, allowing the system to perform rescheduling decisions at each iteration [24]. The *plan switching* approach [121] involves constructing a family of activity graphs beforehand and investigates the means of switching from one member of the family to another when the execution of one activity graph fails, but all the plans are made without knowledge of future environment change. Another rescheduling policy is proposed in [98], which considers rescheduling at a few, carefully selected points during the execution.

2.4.2 Workflow Management Systems

In practice, there have been number of Grid workflow systems developed and evolved either for generic scientific computation or for specific computation domains, including DAGMan/Condor-G [1,22,43], Pegasus [26,89], GrADS [13,24,74], ASKALON [35,36,113], Kepler [10], GridFlow [18], Taverna [83], UNICORE [9, 95, 96], Karajan and Java CoG Kit [63] etc..

When it comes to design and implementation, current workflow management systems tend towards two different extremes [25, 112]:

1. Just-in-time scheduling (in-time local scheduling). The scheduling decision for an individual task is postponed as long as possible, and performed before the task execution starts.

2. Full-ahead planning (workflow planning). The whole workflow is scheduled before its execution starts.

Just-in-time scheduling is represented by many simple scheduling heuristics like Min-min, Max-min, Suffrage and XSuffrage [19, 73, 112, 117].

Among the performance driven workflow management systems, DAGMan [1, 43] and Taverna [83] support dynamic scheduling, GridFlow [18], ASKALON [113] support static scheduling. Pegasus [27] and adaptive scheduling algorithm [122, 123] support both, and therefore hybrid solutions.

2.4.3 Scheduling Multiple Workflow Applications

Additionally, most static algorithms become hardly applicable when it comes to the common reality that a grid environment may have to serve dynamic workload of multiple workflow applications mixed with other independent jobs. All of the 27 static algorithms surveyed in [62] and other ones [92, 98, 111] devised later are restricted to single DAG scheduling. Recent efforts [55, 125] attempt to schedule multiple DAGs in a grid environment, but they

merely merge multiple DAGs into one unified DAG *a priori* and schedule it at one time. In reality, the DAGs may be submitted into a grid environment at different points of time by different users.

On the other hand, dynamic algorithms can handle multiple DAGs in a natural way because of their intrinsic adaptability to the dynamics of both workload and environment. From the scheduler's point of view, a ready to execute individual task within a DAG is no different than other ordinary independent jobs waiting in the queue. As the job interdependence is transparent to the scheduler, it can handle one or many workflow applications.

However, similar to the case of scheduling single workflow, lack of global view in dynamic scheduling is the issue to address in order to improve the performance.

2.4.4 Scheduling with Resource Failures

Resource failure is still very common in any computing system despite the continuous reliability improvement of both software and hardware. Both Open Science Grid [5] and TeraGrid [4] report more than 30% failures at times [119]. Noticeable progress has been made on failure prediction research and practice, following that more failure traces are made publicly available since 2006 and the failure analysis [45, 68, 87, 101, 118, 124] reveals more failure characteristics in high performance computing systems. Zhang *et al.* evaluate the performance implications of failures in large scale cluster [124]. Fu *et al.* propose both online and offline failure prediction models in coalitions of clusters. Another failure prediction model is proposed by Liang *et al.* [68] based on failure analysis of BlueGene/L system. Recently, Ren et al. [94] developed a resource failure prediction model for fine-grained cycle sharing systems. However, most of them focus on improving predication accuracy, and few of them study how to leverage their predication results in practice.

Salfner *et al.* [100] suggest that proactive failure handling provides the potential to improve system availability up to an order of magnitude, and the FT-Pro project [67] and the FARS project [66] demonstrate a significant performance improvement for long-running

applications provided by proactive fault tolerance policies. Fault aware job scheduling algorithms are developed for BlueGene/L system and simulation studies show that the use of these new algorithms with even trivial fault prediction confidence or accuracy levels (as low as 10%) can significantly improve the system performance [84].

Failure handling is considered in some workflow management systems but only limited in failure recovery. Grid Workflow [60] presents a failure tolerance framework to address the Grid-unique failure recovery, which allows users to specify the failure recovery policy in the workflow structure definition. Abawajy [7] proposes a fault-tolerant scheduling policy that loosely couples job scheduling with a job replication scheme such that applications are reliably executed but with a cost of resource efficiency. Other systems such as DAGMan [1] simply ignore the failed jobs, having the job rescheduled later when it is required for dependant jobs. Dogan *et al.* [29] develop Reliable Dynamic Level Scheduling (RDLS) algorithm to factor resource availability into conventional static scheduling algorithms.

We argue that, however, failure handling can not be practically integrated into existing static scheduling schemes as it is not possible to predict all failures accurately in advance for a long running workflow application. Even for other non-workflow applications, the analysis [87] finds that node placement decision can become ill-suited after about 30 minutes in a shared federated environment such as PlanetLab [90]. Furthermore, another analysis [118] concludes that time to fail (TTF) and time to repair (TTR) can not be predicted with reasonable accuracy based on current uptime, downtime, mean time to fail (MMTF) or mean time to repair (MMTR) and a system should not rely on such predictions.

2.4.5 Scheduling Workflows in Multicluster Environments

Since the early stage of Grid when heterogeneous workstations and PCs were interconnected to provide a more powerful distributed computing platform, incredible progresses have been made on supporting security and resource access across domains, such as Globus Toolkit

Version 4 [38], clusters are connected with high speed backbone to build explosively more powerful Grids, such as TeraGrid [4], Planet Lab [90] etc..

In response to increasing number of available resources in large scale distributed environments, an empirical study [56] is performed on automatically generating resource specification for workflow applications in order to get the "best" resource collection. But the work is still at an early stage.

While most DAG scheduling heuristics in literature assume that each computational resource is a processing node of a single processor, the study [51] demonstrates that if the resource is actually a cluster with multiple processing nodes, this assumption will cause a misconception in the tasks' execution time and execution order which further adversely impact the scheduler efficiency.

Recently, RePA [57] is proposed as a dynamic scheduling algorithm to reduce the communication and redistribution costs by mapping child tasks to processors which are assigned to parent tasks(reuse processors). However, the algorithm makes a fundamental assumption that tasks are always mapped to single cluster and thus there is no communication cost for those processors which are shared between the parent and the child task. This assumption can not be true in most cases.

Another algorithm, DMHEFT [58], uses "postponing" approach to schedule a workflow onto multicluster. The heuristic takes care of unfavorable placements of multiprocessor task by considering the postponing of ready tasks even if idle processors are available. The scheduler postpones the task if it thinks more resource will be available after a certain mount of wait time. However, with the assumption that resource is dedicated for the workflow, this heuristic does not consider that a task submitted to the cluster has to compete with other jobs already in the queue.

To consider the resource competition in the real world scenario, it is crucial to be able to predict queue wait time for a given job. A breakthrough prediction approach named QBETS [80, 81] is recently developed and deployed on more than a dozen super computing

sites, offering on-line queue delay predictions for individual jobs. The study [82] evaluates a workflow scheduler integrated with prediction approach.

CHAPTER 3
WORKFLOW SCHEDULING SYSTEM DESIGN

This chapter presents the conceptual system architecture of a proposed planner guided dynamic scheduling system in Section 3.1, followed by Section 3.2 and Section 3.3 which describe the concepts of local priority and global priority respectively.

3.1 Conceptual System Architecture

As discussed in the introduction of workflow scheduling research background, Section 2, current workflow managements are designed in two extremely different ways. The scheduling decision is either made by the workflow *Planner* before execution, i.e. static scheduling, or by the *Executor* just in time, i.e. dynamic scheduling.

The advantage of static scheduling comes from the prior knowledge that how each task's execution time impacts on overall workflow performance so that critical tasks are mapped to the best suitable resource collection. However, in a grid environment static strategies may perform poorly because what actually happens during execution can be totaly different than what is envisioned at plan time due to the grid dynamics.

On the other hand, dynamic scheduling can manage the dynamic resources and workloads as the decision is only made when a task is ready to execute. This type of decision is also referred to as local just-in-time decision. However, from perspective of performance, it is widely believed that static strategies can outperform dynamic ones [14, 113].

We propose a system design which is a hybrid solution and adapts the *Planner* to dynamic grid environment via collaboration with the *Executor*, as shown in Fig. 3.1.

The proposed system consists of four core components: *DAG Planners*, a *Job Pool*, an *Executor*, an *Online Failure Predictor*. The fabric of the system is the *GRID Services* which

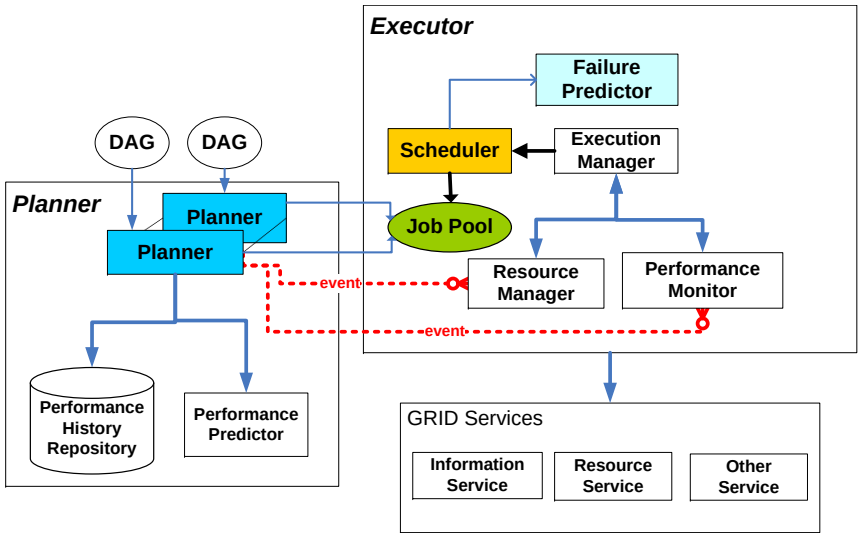


Figure 3.1: Conceptual system architecture.

is a collection of essential services provided by any grid system and thus not the focus of this dissertation.

The *DAG Planner* assigns each individual task a local priority by employing selected static heuristic, manages the task interdependence and submit ready to execute tasks to the *Job Pool*, which is an unsorted collection containing all ready to execute jobs from different users. The *Executor* re-prioritizes the jobs in the *Job Pool* and schedules tasks to the available resources in the order of job priority. When making a scheduling decision, the *Executor* will consult the *Failure Predictor* about whether a resource will keep alive for the entire task execution period if the task is assigned to this resource. If a task is terminated due to unpredicted resource failure, the *Executor* will place the task back into *Job Pool* and the task will be rescheduled. When a task finishes successfully, the *Executor* notifies the *DAG Planner* which the task belongs to of the completion status.

With the collaboration introduced in this design, the *Planner* can utilize the prior knowledge of DAG and performance prediction to make resource mapping in favor of overall workflow application performance. However, during workflow enactment, the *Planner* only

submits the ready to execute jobs to the global *Job Pool* managed by *Executor* along with the local priority and resource mapping information. It will be the responsibility of the *Executor* to make final scheduling decisions based on real time system status. This approach makes system adaptive to dynamics of both Grid systems and workloads, the above collaboration among these core components is achieved by the dynamic event driven design illustrated in Fig. 3.1 and explained as follows:

1. *Job submission.* When a new DAG arrives, it is associated with an instance of *DAG Planner* by the system. After ranking all individual jobs within the DAG locally, the *Planner* submits whichever job is ready to the *Job Pool*. At the beginning, only entry job(s) will be submitted. Afterwards, upon notification by the *Executor* of the completion of a job, the *Planner* will determine if any dependant job(s) become ready and submit them. During the course of workflow execution, the job terminated due to resource failure is put back to the *Job Pool* by the *Executor* to be rescheduled later. In other words, the *Planner* maintains a global view of the workflow which it is associated with, determines local priorities of individual tasks, monitors the job completion status and submits the jobs which are ready to execute. The *Planner* can adopt any predefined ranking mechanism used in list based heuristics.

2. *Job scheduling.* Whenever there are resources available and a job is waiting in the *Job Pool*, the *Executor* will repeatedly do:

   (a) Re-prioritize all jobs residing in the *Job Pool* based on individual job ranks in a real time fashion.

   (b) Pick up the job with the highest global priority from *Job Pool* to schedule;

   (c) Schedule the job to the resource which allows the earliest finish time and will not fail during job execution.

   The detailed job scheduling policy is discussed in the subsequent chapters from various perspectives.

3. *Job completion notification.* When a job finishes successfully, the *Executor* will notify the corresponding *DAG Planner* of job completion status.

3.2 Task Ranking and Local Priority

In the system design, the *Planner* prioritizes each individual task with a ranking mechanism introduced by Heterogenous Earliest-Finish-Time(HEFT) heuristic [111].

HEFT is one of the most popular heuristics, implemented in the grid project ASKALON [113] and proven superior to other alternatives. Some other heuristics are studied in a comprehensive evaluation [54], and surprisingly they show a very similar behavior regarding the quality of the obtained results, exhibiting the same strengths and weaknesses, differing only by few percent. Based on these observations, *HEFT* heuristic is selected in the design as a ranking mechanism.

The rank value of all tasks are computed recursively, starting from the exit task(s) of a workflow. Following the definitions 2.1 and 2.3 in workflow scheduling modeling in Section 2.1, the rank value of task n_{exit} is defined as

$$rank_u(n_i) = \overline{w_{exit}} \tag{3.1}$$

For other non-exit tasks, the rank value is computed recursively defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{(i,j)}} + rank_u(n_j)) \tag{3.2}$$

where $succ(n_i)$ is the set of immediate successors of task n_i, $\overline{c_{i,j}}$ is the average communication cost of edge(i, j), and $\overline{w_i}$ is the average constatation cost of task n_i.

Basically, the $rank_u(n_i)$ is the length of the critical path from task n_i to the exit task, including the computation cost of task n_i. It also guarantees that the rank value of n_i is always higher than any successor and the entry task(s) have the highest values. Intuitively, scheduling the task with higher rank earlier naturally observes the intra-task dependency.

Furthermore, as the task with higher rank is more critical than ones with lower one, it should be assigned with the better resource collection to minimize the total execution time.

3.3 Global Priority

In reality, the workload in a computing system is dynamic and mixed of various application types. Tasks in the *Job Pool* can come from different users and different applications. It is usual in any typical batch queue system that some tasks are from various workflows while others are ordinary independent jobs.

One of the primary responsibilities of the *Executor* is to enforce a good scheduling policy which considers the *resource utilization efficiency* and *fairness*, in addition to workflow performance. Existing heuristics assume only a single workflow in the environment and does not consider any resource competition, let alone *fairness*.

Therefore, the *Executor* re-assigns the tasks in the *Job Pool* with global priorities with each task's local priority as one of the inputs. The task with higher priority is scheduled earlier with more choices of resource collections, similar to single workflow scheduling situation. With the re-assigned global priorities, the tasks from same DAG still keep same order as they are locally. This helps achieve high application performance for the workflow. In addition, the tasks from different applications(users) are ordered in a dynamic way to ensure the *fairness* for each application and avoid resource starvation. A global prioritization approach is proposed in Chapter 5 to schedule multiple workflows.

3.4 Summary

In this chapter, we provide an overview of the system architecture for the propose hybrid planner guided multi-workflow scheduling in cluster and grid environment. We further describe each primary component and its functionality, and how they work collaboratively to support the scheduling strategy. The subsequent chapters will elaborate how the scheduling strategy manage dynamic resource and dynamic workload in details.

CHAPTER 4
ADAPTIVE SCHEDULING

Even though theoretically static scheduling performs near to optimal, its practicability and effectiveness in a dynamic grid environment is always questioned. We discuss and analyze these issues in Section 4.1, and propose in Section 4.2 a static strategy based *adaptive rescheduling* algorithm by which the workflow *Planner* can adapt to the grid dynamics to realize its strength practically. The experiment design and results are presented in Section 4.3 showing that the proposed strategy not only outperforms the dynamic ones but also improves over the traditional static ones.

4.1 Issues with Static Scheduling

Planning is a one time activity in the traditional static scheduling paradigm. It does not consider the future dynamic change of grid environment after the resource mapping is made.

However, in a grid environment static strategies may perform poorly because of the grid dynamics: resource can join and leave at any time; individual resource capability varies over time because of internal or external factors; and it is not easy to accurately estimate the communication and computation cost of each task, which is the foundation of any static scheduling.

Overall, the issues with traditional static scheduling are: (1) *Accuracy of estimation.* Estimating communication and computation costs of a DAG is the key success factor but practically difficult. The deviation in run time is detrimental to scheduling based scheduling. (2) *Adaptation to dynamic environment.* Most static scheduling approaches assume that resource set is given and fixed over time. The assumption is not always valid even with the reservation capability in place. Moreover, the static scheduling approach can not utilize

new resources after the plan is made; and (3) *Separation of workflow planner from executor*. Fundamentally the above two issues are related to the lack of collaboration between the workflow planner and executor. With collaboration, a planner will be aware of the grid environment change, including the job performance variance and resource availability, and is able to adaptively reschedule based on the increasingly accurate estimations. This approach can both continuously improve performance by considering the new resources and minimize the impact caused by unexpected resource downgrade or unavailability.

We argue that the promising benefits of static strategies can be practically realized with collaboration between workflow planner and executor, which is currently missed in most system designs. We propose an *HEFT* [111] based adaptive rescheduling algorithm to support such desired collaboration. With this approach, the executor will notify the planner of any run time event which interests the planner, for example, resource unavailability or discovery of new resource. In turn, the planner responds to the event by means of evaluating the event and rescheduling the remaining tasks in the workflow if necessary. Planning is now an iterative (event-driven) activity instead of one time task. The experiment results, including simulation on both parametric randomly generated DAGs and two real application DAGs, show a considerable performance improvement by adaptive rescheduling.

4.2 An Adaptive Scheduling Algorithm

We present the basic idea of adaptive scheduling in Section 4.2.1, followed by a detailed algorithm based on HEFT [111] in Section 4.2.2.

4.2.1 Adaptive Rescheduling

For a given DAG and a set of currently available resources, the *Planner* makes the initial resource mapping as any other traditional static approaches do. The primary difference is that our approach requires the *Planner* listens for and adapts to the significant events in the execution phase, such as:

- *Resource Pool Change.* If new resource is discovered after the current plan is made, rescheduling may reduce the *makespan* of a DAG by considering the resource addition. When resource fails, fault tolerant mechanism is triggered and it is taken care of by the *Executor*. However, if the failure is predictable, rescheduling can minimize the failure impact on overall performance.

- *Resource Performance Variance.* The performance estimation accuracy is largely dependent on history data, and inaccurate estimation leads to a bad schedule. If the run time *Performance Monitor* can notify the *Planner* of any significant performance variance, the *Planner* will evaluate its impact and reschedule if necessary. In the meantime, the *Performance History Repository* is updated to improve the estimation accuracy in the subsequent planning.

The *Planner* reacts to event by evaluating if *makespan* can be reduced by rescheduling. For example, if a new resource becomes available, the *Planner* will evaluate if a new schedule with the extra resource in consideration can produce smaller *makespan*. If so, the *Planner* will replace the current one with new one by submitting it to the *Executor*.

The evaluation can be further extended to support online system management function by answering the "*What...if...*" type query, for example, "*What* will be the expected performance *if* an additional resource A is added (removed)?" The query result, as evaluation output, will help one to tune up the application and system performance in a proactive way, and this will be our future work.

A generic adaptive rescheduling algorithm is described in Fig. 4.1. For a given DAG, initially or when an event occurs during its execution, the *Planner* schedules or evaluates the event by (re)scheduling. The *Planner* retrieves the latest resource information and job performance history data first, and estimates the cost of each job in the DAG. Based on the estimation, the *Planner* applies a specific static heuristic, for example *HEFT*, either makes an initial schedule for the entire DAG or a new schedule for the remaining tasks. If the schedule is an initial one or it is expected to perform better than the current one, the

Planner submits it to the *Executor* to execute, otherwise the *Planner* does not take any action. Until the DAG is executed successfully, the *Planner* keeps listening for the event of interest, evaluate it and reschedule the DAG if necessary. S_0 and S_1 denotes the current schedule and new one respectively.

```
    T - set of the tasks in the DAG
    R - set of all available resources
    P - performance estimation matrix
    H - Heuristic employed by scheduler
    S - Schedule
1.  set initial schedule S0 = null
2.  while ((S0==null OR any event) AND DAG  not finished) do
    #R is updated via the communication with Resource Manager
3.  update Resource Set R;
4.  update Performance History Repository;
    #Predicator component will update performance estimation matrix P
5.  call P=estimate(T, R);
    #New schedule is made by applying the heuristics H on execution
        status snapshot of S0 and P
6.  call S1=schedule(S0, P, H);
7.  if (S0==null OR S0.makespan>S1.makespan)
8.      S0=S1;
9.      submit S0;
10. endif
11. endwhile
```

Figure 4.1: A generic adaptive rescheduling algorithm.

4.2.2 HEFT-based Adaptive Rescheduling: AHEFT

Next we define our own adaptive scheduling strategy, which is an *HEFT*-based adaptive rescheduling algorithm, referred to as *AHEFT* hereafter. Specifically, we use *HEFT* heuristic to implement the $schedule(S_0, P, H)$ method in the generic algorithm described in Fig. 4.1. We model AHEFT based on the the workflow model defined in Section 2.1 and extended from [111] with revision. A workflow application is represented by a direct acyclic graph, *G=(V, E)*, where V is the set of v tasks (nodes) and E is the set of e edges between

Table 4.1: Definition of attributes in *AHEFT*

Attribute	**Definition**
$EST(n_i, r_j, S_0, clock, R)$	the earliest start time for not-started task n_i on resource r_j with available resource set R when the schedule S_0 is executed to the time point of *clock*
$EFT(n_i, r_j, S_0, clock, R)$	the earliest finish time for not-started task n_i on resource r_j with available resource set R when the schedule S_0 is executed to the time point of *clock*
$FEA(n_m, n_i, r_j, S_0, clock)$	the earliest time for file output of task n_m being available on resource r_j ready for task n_i after schedule S_0 has been executed to the time point of *clock*
$SFT(n_i)$	scheduled finish time of task n_i when it is mapped to a resource
$AST(n_i)$	actual start time of task n_i
$AFT(n_i)$	actual finish time of task n_i
$avail[j]$	the earliest time when resource r_i is ready for task execution
$w_{i,j}$	the computational cost of task n_i on resource r_j
$c_{i,j}$	the communication cost for data dependence of task n_j on n_i
$pred(n_i)$	the set of immediate predecessor tasks of task n_i

tasks. Each edge $(i, j) \in E$ represents the precedence constraint such that task n_i should complete its execution before task n_j starts. *data* is a $v \times v$ matrix of communication data, where $data_{i,k}$ is the amount of data required to be transmitted from task n_i to task n_k. R is a set of resources which represent computation units. The variable *clock* is used as logical clock to measure the time span of DAG execution, it is initially set as 0 before the DAG starts to execute. When the DAG finishes successfully, the *clock* reads as the *makespan* of the DAG.

we define the symbols used by *AHEFT* in Table 4.1, and formulate the calculation of these attributes by these three equations, whose rationale are described next.

$$FEA(n_m,n_i,r_j,S_0,clock)=\begin{cases} AFT(n_m), & Case1 \\ clock+c_{m,i}, & Case2 \\ SFT(n_m), & Case3 \\ SFT(n_m)+c_{m,i}, & \text{otherwise.} \end{cases} \tag{4.1}$$

wherein,

Case 1: If task n_m finished on resource r_j;

Case 2: If task n_m finished but its output is not scheduled to transfer to resource r_j.

Case 3: If task n_m has not finished **and** is mapped to resource r_j in new schedule.

$$EST(n_i,r_j,S_0,clock,R)= \\ max\{avail[j],\max_{n_m\in pred(n_i)}(FEA(n_m,n_i,r_j,S_0,clock))\} \tag{4.2}$$

and

$$EFT(n_i,r_j,S_0,clock,R)=w_{i,j}+EST(n_i,r_j,S_0,clock,R) \tag{4.3}$$

A task can not start without all required inputs being ready on the resource on which the task is to execute. Such inputs are in turn the outputs from immediate predecessor tasks. If a task n_i will execute on resource r_j and requires output data from an immediate preceding task n_m, the Equation (4.1) calculates the earliest time when output data arrives on resource r_j. By the time of (re)scheduling, *clock*, if task n_m already finishes on resource r_j then its output is ready there as input for task n_i, and the FEA equals $AFT(n_m)$. If task n_m finishes but on different resource, then its output has to transfer to resource r_j. As the

file transmission can not be earlier than *clock*, the FEA is equal to $clock + c_{m,i}$. Otherwise, if a task n_m is not finished or its output is not transferred to resource r_j by *clock*, it will be rescheduled in new schedule S_1.

On the other hand, a task can not execute before the earliest available time $avail[j]$ for resource r_j. These constraints are indicated by the inner *max* block in Equation (4.2). It is easy to get the earliest finish time of task n_i by adding the estimated execution time $w_{i,j}$ to its earliest start time. After a task n_i is scheduled on resource r_j, the earliest finish time for task n_i on resource r_j is denoted as $SFT(n_i)$, the scheduled finish time of task n_i. Finally the *makespan* is defined as,

$$makspan = max\{SFT(n_{exit})\} \tag{4.4}$$

where n_{exit} is the exit task in a DAG. There can be one or multiple exit tasks in one DAG.

It is obvious that *AHEFT* is identical to *HEFT* [111] when $clock = 0$ or it is the initial scheduling, i.e. S_0 is not defined yet. The primary difference comes to the rescheduling when *AHEFT* considers the fact that workflow has been executed partially. In Equation (4.1), the FEA can be actual available time if by the time of rescheduling, i.e., *clock*, the immediate predecessor task finishes and output file is moved or to be moved to the resource as previously scheduled. However, previous schedule may direct the output file to different resource, then the file needs to be retransmitted to this resource regardless, which falls into the second situation in Equation (4.1). The third one is the same as *HEFT*, if either this is initial scheduling or the immediate predecessor task has not started yet. With these equations now we can define the procedure $schedule(S_0, P, H)$ of *AHEFT*, see Fig. 5.4.

Except for how EFT is calculated, the procedure $schedule(S_0, P, H)$ defined in Fig. 5.4 is very similar to *HEFT*. Based on the cost estimation obtained, i.e., line 5 in Fig. 4.1, the upward rank of a task n_i is recursively computed as defined in Section 3.2, starting from the task n_{exit}.

```
   T - set of the tasks of status not started in DAG
   R - set of all available resources
   P - performance estimation matrix
   H - HEFT heuristic employed by scheduler
   S0 - Initial schedule
   clock - the time point of scheduling
1. procedure schedule(S0, P, H)
2.    compute ranku for all tasks by traversing graph upward, starting
        from the exit task
3.    sort the tasks in a scheduling list by nonincreasing order of ranku
4.    while there are unscheduled tasks in the list do
5.       select the first task, ni from the list of scheduling
6.       for each resource rk in R do
7.          compute EFT(ni, rk , S0, clock, R)
8.          assign task ni to the resource rj that minimizes EFT of task ni
9.    endwhile
```

Figure 4.2: Procedure $schedule(S_0, P, H)$ of $AHEFT$.

As indicated by line 2 and 3 in Fig. 5.4, the upward rank is calculated for each task and sorted in nonincreasing order which corresponds to significance order how the individual task affects the final *makespan*. The basic concept of this algorithm is to select the "best" resource which minimizes the earliest finish time of the task currently with highest upward rank and remove the task from unscheduled task list once it is assigned with resource. The resource selection process repeats until the list is empty.

As an illustration, we use a sample DAG and resource set, shown in Fig. 2.1 of Section 2.1, to compare schedule performance of traditional $HEFT$ and $AHEFT$. Fig. 4.3 shows the schedule obtained from traditional $HEFT$ and $AHEFT$ respectively. Resources r_1, r_2 and r_3 are available from the beginning, while resource r_4 emerges at time point of 15. $HEFT$ produces the schedule with makespan as 80 without considering the addition of resource r_4 at later time. For $AHEFT$, the initial schedule made at time point of 0 is identical as the one by $HEFT$. However, when resource r_4 is added, $HEFT$ reschedules the rest of the workflow,

i.e. all tasks but n_1 which is finished by the time of rescheduling. The new schedule reduces the *makespan* to 76.

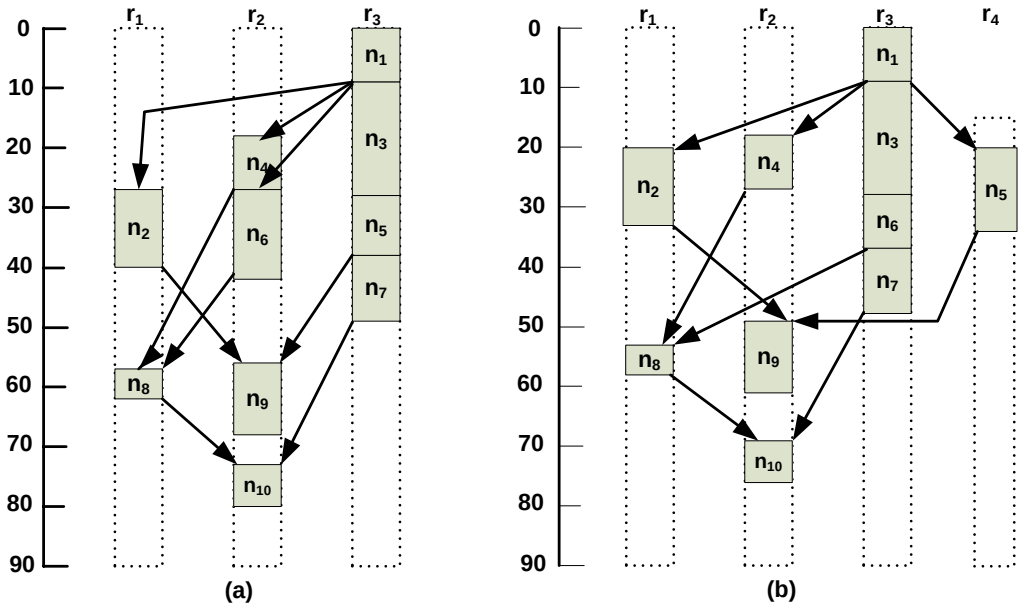


Figure 4.3: Schedule of the DAG in Fig. 2.1 using *HEFT* and *AHEFT* algorithms: (a) *HEFT* schedule (*makespan*=80) and (b) *AHEFT* schedule with resource adding at time 15 (*makespan*=76).

4.3 Experiment Design and Results

In this section, we present the experiment design for evaluating the effectiveness of *AHEFT*. We first evaluate it with randomly generated DAGs. Then we specifically compare it with traditional *HEFT* in the context of two real world applications, namely BLAST [106] and WIEN2K [114].

4.3.1 Experiment Design

The following important assumptions are made for the experiment design: (1) *Accuracy of estimation.* As other studies [14,98,111], the estimation of communication and computation cost is assumed accurate and task will start and finish on time; (2) *File transferring.* For

static approaches, when a task finishes, the output file of the task is transmitted immediately to the resources where the immediate succeeding tasks are scheduled to execute on. But for dynamic one the output file is not transmitted until the *Executor* decides on which resource to run the depending task. In both cases, the file transmission is time consuming only activity and does not incur computation cost; and (3) *Advance resource reservation.* We assume the advance reservation capability ensures resource availability during the reserved time window. On the other hand, *HEFT* and *AHEFT* react identically to the resource failure while task is executing, as if rescheduling is the fault tolerance mechanism. Therefore, to simplify the experiment design, we can reasonably only consider the situation that new resources come available during the execution of workflow.

4.3.2 Results of Parametric Randomly Generated DAGs

In order to evaluate the performance and stability of *AHEFT*, i.e., whether it always performs better than *HEFT* and dynamic one in all kinds of cases, we use parametric randomly generated DAGs in the experiment. For the purpose of fair comparison, we directly follow the heterogeneous computation modeling approach defined in [111] to generate representative DAG test cases. The input parameters and the corresponding values are very similar as used in [111] as well. These input parameters are also suggested in the workflow test bench work [54], as listed below:

- The number of tasks in the graph (v).
- The maximum out edges of a node, out_degree, represented as percentage of total nodes in a DAG.
- Communication to computation ratio (CCR). A data-intensive application has a higher CCR, while a computing-intensive one has a lower value.
- The resource heterogenous factor, β. A higher value of β suggests the bigger difference of resource capability. The resources are homogeneous when β is 0. The average

computation cost of all tasks in a DAG is $\overline{\omega_{DAG}}$, then the average of each task n_i in the graph, represented as $\overline{\omega_i}$, is selected randomly from a uniform distribution with range $[0, 2 \times \overline{\omega_{DAG}}]$. Then, the computation cost of each task n_i on each resource r_j in the system, i.e., $\omega_{i,j}$, is randomly selected from the following range: $\overline{\omega_i} \times (1 - \frac{\beta}{2}) \leq \omega_{i,j} \leq \overline{\omega_i} \times (1 + \frac{\beta}{2})$.

Table 4.2: Parameter values of random generated DAGs.

Parameter	**Value**
v	20, 40, 60, 80, 100
CCR	0.1, 0.5, 1.0, 5.0, 10.0
out_degree	0.1, 0.2, 0.3, 0.4, 1.0
β	0.1, 0.25, 0.5, 0.75, 1.0
R	10, 20, 30, 40, 50
Δ	400, 800, 1200, 1600
δ	0.10, 0.15, 0.20, 0.25

To model the dynamic change of resources, we introduce three additional parameters as following: (1) Initial resource pool size, R; (2) Interval of resource change, Δ. The higher value of Δ indicates the lower frequency of resource change; and (3) Percentage of resource change, δ, to measure the resource change percentage each time compared with the initial resource pool. The value set for each parameter of this empowerment is listed in Table 4.2.

With combination of v, CCR, out_degree and β, we have totally 625 different DAG types. For each type we create 10 instances with randomly assigned computation and communication cost, so there are totally 6250 DAGs used in the experiment. Then we apply 80 different types of resource models, combining the R, Δ and δ, so we finally generate 500,000 test cases. For each DAG, we simulate *HEFT* [111], *AHEFT* and dynamic *Min-Min* [42] heuristic and obtain the respective *makespan*. The simulation for dynamic *Min-Min* is implemented on top of the event-driven simulation framework SimJava [6].

The average *makespan* for *HEFT*, *AHEFT* and *Min-Min* are 4075, 3911 and 12352 respectively. It shows that both *HEFT* and *AHEFT* achieve much better performance than *Min-Min*, and *AHEFT* is slightly better than *HEFT*. We further compare *AHEFT* and

Table 4.3: Improvement rate with various *CCR*s.

CCR	0.1	0.5	1.0	5.0	10.0
Imprv. rate	0.4%	0.5%	0.7%	3.2%	7.7%

Table 4.4: Improvement rate with various total number of tasks.

Task number	20	40	60	80	100
Imprv. rate	2.9%	3.9%	4.3%	4.2%	4.1%

HEFT to identify which type of workflow applications can benefit more from *AHEFT* by studying the effect of different parameters. Given the limited space, we show the results of CCR and the number of tasks in Table 4.3 and Table 4.4 respectively. One can easily notice that *AHEFT* favors data-intensive workflow application by Table 4.3. When CCR increases, i.e., application is more data-intensive, *AHEFT* outperforms *HEFT* better. Another observation is, with the total number of tasks increases, the improvement rate jumps initially and becomes stable later, as Table 4.4 shows.

It is worth noting that these observations are drawn from the experiments with randomly generated DAGs of limited scale (less or equal to 100 tasks). To better understand the correlation between *AHEFT* and workflow application characteristics, we evaluate with two real world applications in the next subsection.

4.3.3 Results of BLAST and WIEN2K

We attribute the less significance of the performance improvement in randomly generated DAGs to two observations below: 1) *DAG shape*. Typically a scientific workflow application is designed to accomplish a complex task by means of job parallelism, its DAG is hence uniquely shaped. The DAGs of many real world workflow applications are well balanced and highly parallel, like Montage [3], BLAST [106] and WIEN2K [114], and so forth. Moreover, the DAG shape decides the job parallelism degree to some extent; 2) *Types of tasks in the DAG*. Despite of the fact that one scientific workflow is composed of hundreds individual

tasks if not thousands, there are only handful unique operations. For example, Montage has totally 11 unique executable operations. The same operation appears as different individual tasks in the DAG when it is executed in different context with different inputs. This observation holds same true with BLAST and WIEN2K applications. Fig. 4.4 gives a six-step BLAST workflow example with two-way parallelism. This workflow represents a set of function calls that specify inputs such as genome sequence files, output files from comparative analysis tools, and textual parameters. We conduct the simulation with 200-, 400-, 600-, 800- and 1000-way parallelism respectively. With these two observations we choose BLAST and WIEN2K DAGs to evaluate how well adaptive rescheduling may improve practically and how its effectiveness is related to the DAG characteristics. BLAST and WIEN2K are implemented in grid system GNARE [106] and ASKALON [113] respectively.

WIEN2k [114] is a quantum chemistry application developed at Vienna University of Technology. WIEN2k workflow contains two parallel sections $LAPW1$ and $LAPW2$, with possibly multiple parallel tasks. The DAG we used for experiment is a full-balanced graph, with equal number of parallel tasks in these two sections, as shown in Fig. 4.5. In the experiment, we set the number of parallel tasks as 200, 400, 600, 800, 1000 respectively. The parallelism factor used in both BLAST and WIEN2K actually decides the total number of tasks in the DAG.

We define the value set for each parameter of experiment with both BLAST and WIEN2K in Table 4.5. Table 4.6 shows the average *makespan* improvement by $AHEFT$ over

Table 4.5: Parameter values of BLAST and WIEN2K DAGs.

Parameter	**Value**
υ	200, 400, 600, 800, 1000
CCR	0.1, 0.5, 1.0, 5.0, 10.0
β	0.1, 0.25, 0.5, 0.75, 1.0
R	20, 40, 60, 80, 100
Δ	400, 800, 1200, 1600
δ	0.10, 0.15, 0.20, 0.25

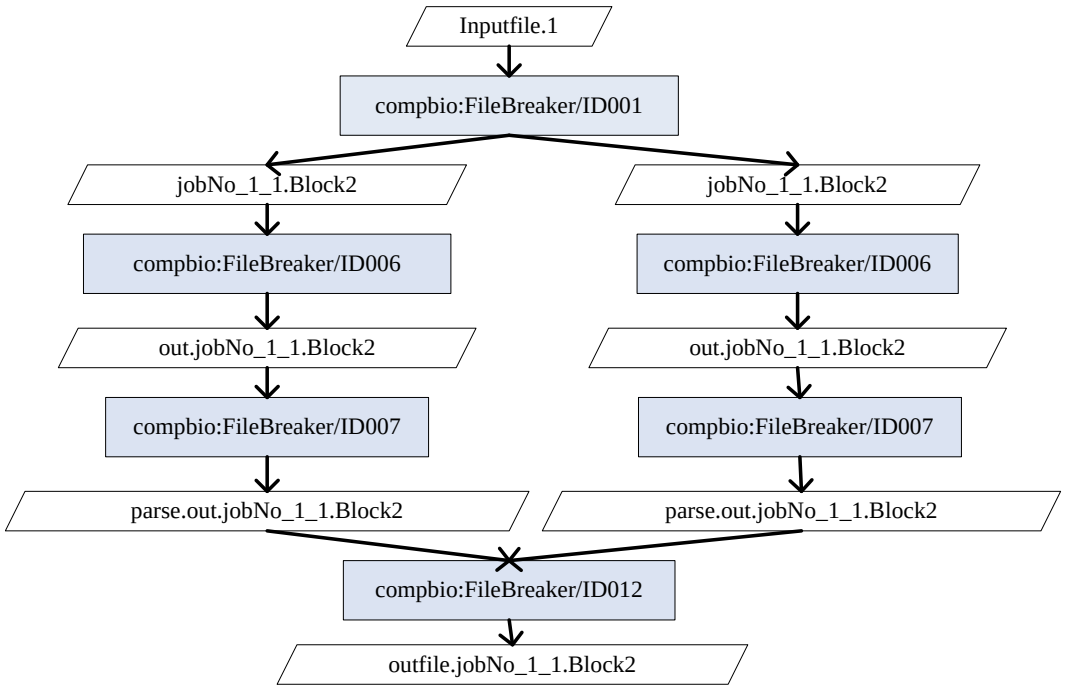


Figure 4.4: A six-step BLAST workflow with two-way parallelism [106]. The rectangle represents a task and the parallelogram represents data file.

Table 4.6: Average *makespan* and improvement rate by *AHEFT*.

Application	***HEFT***	***AHEFT***	**Improvement rate**
BLAST	4939.3	3933.1	20.4%
WIEN2K	3451.6	3233.8	6.3%

BLAST and WIEN2K respectively. The results again assert that the effectiveness of adaptive rescheduling is very sensitive to the parallelism degree of DAGs, which in turn relates to the DAG shape, well corresponding to our observation mentioned earlier of this section. For the DAGs with shape like BLAST, *AHEFT* can help to reduce *makespan* by 20.4% on average when new resources are added to the system periodically. But it only improves a little with the WIEN2K DAG. The difference is understandable if one notices that the parallelism degree of WIEK2K is obviously lower than that of BLAST, so that any additional resource is less likely utilized and contributes less to the performance improvement. Despite

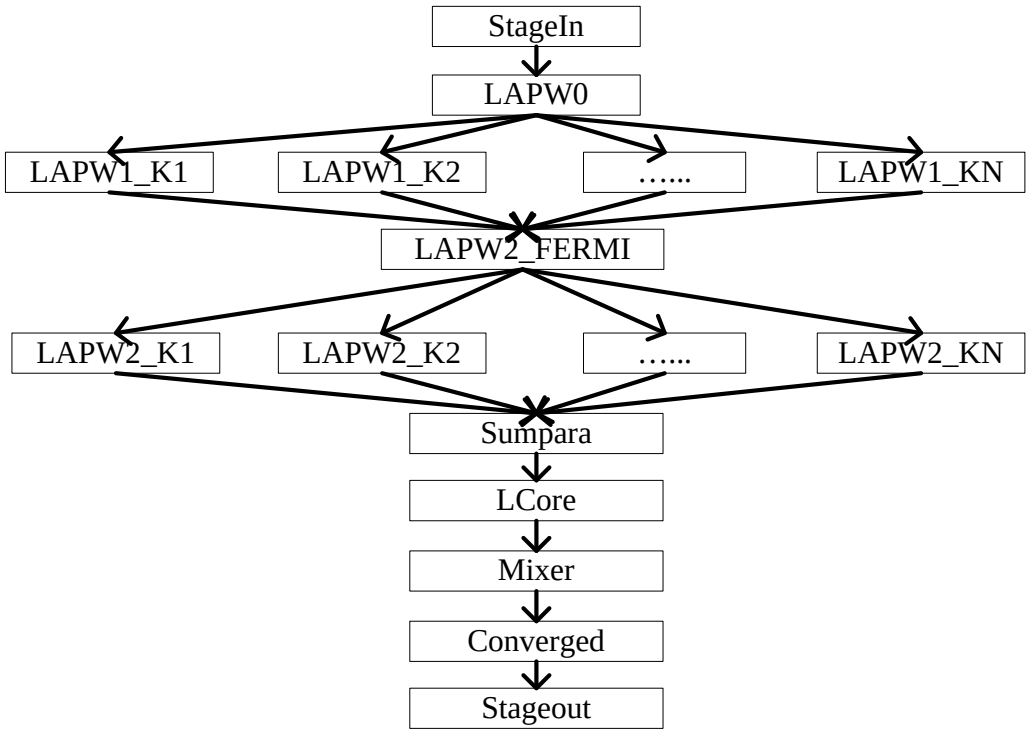


Figure 4.5: A full-balanced WIEN2K DAG example [113].

of the high parallelism degree in each of two sections (*LAPW1* and *LAPW2*) of WIEN2K DAG, the task *LAPW2_FERMI* is the single task on its level, which reduces the parallelism significantly because any task in the *LAPW2* section can not start until this task finishes. It is easy to conceive that extra resources can not help a single task if it can only utilize one resource at a time, which leaves other available resources idle.

We further study the correlation between the performance improvement rate and DAG parameters and show them in Fig. 4.6 from six perspectives: (a) Relationship of *makespan* and CCR; (b) Relationship of *makespan* and β; (c) Relationship of *makespan* and total number of tasks; (d) Relationship of *makespan* and initial resource pool size; (e) Relationship of *makespan* and resource change frequency and (f) Relationship of *makespan* and resource change percentage. To better illustrate how *AHEFT* improves the schedule variously with different DAG parameters, Fig. 4.6 presents the results of *HEFT* and *AHEFT* for both

BLAST and WIEN2K, where HEFT1 and AHEFT1 represent application of *HEFT* and *AHEFT* on BLAST respectively, similarly HEFT2 and AHEFT2 represent for application of *HEFT* and *AHEFT* on WIEN2K respectively. The improvement rate increases as the DAG

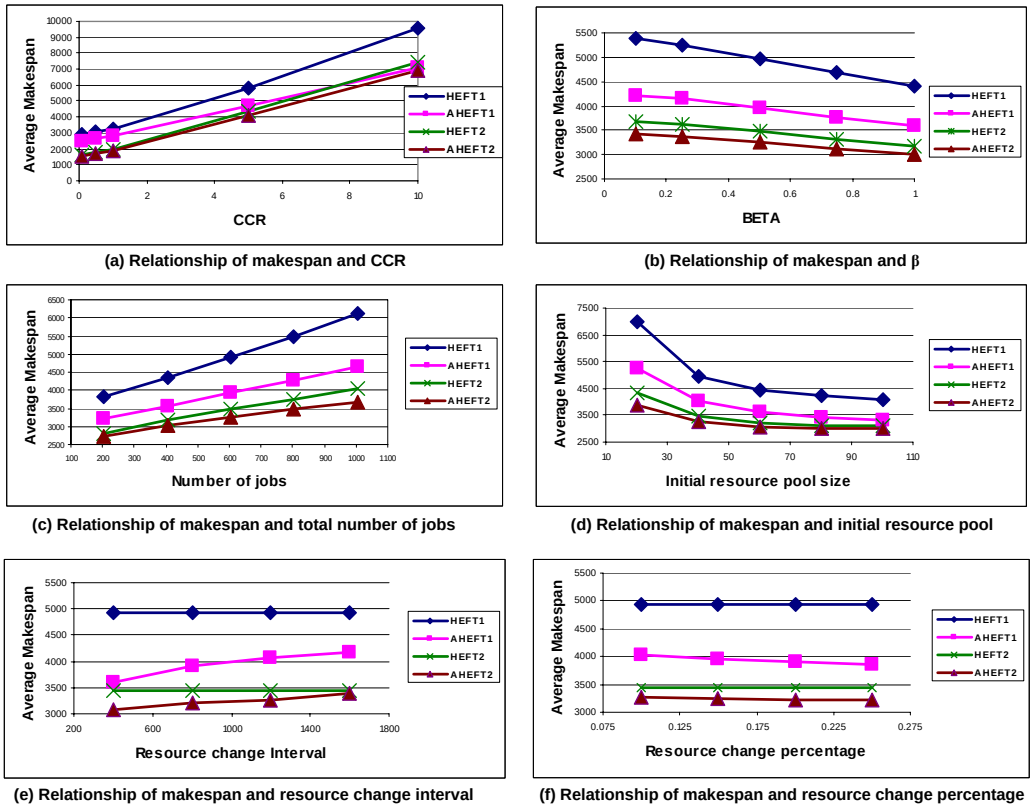


Figure 4.6: Relationship of average *makespan* and different parameters. **HEFT1**: applying *HEFT* on BLAST DAG, **AHEFT1**: applying *AHEFT* on BLAST DAG, **HEFT2**: applying *HEFT* on WIEN2K DAG, **IHEFT2**: applying *AHEFT* on WIEN2K DAG.

gets more complex, i.e., the total number of tasks gets bigger, as Table 4.7 and Fig. 4.6(c) show. This holds true for both BLAST and WIEN2K applications, and the rate accelerates faster with WIEN2K than BLAST. It implies that adaptive rescheduling is more effective for more complex DAGs. When *CCR* goes up, the improvement rate increases slightly as well, as shown in Fig. 4.6(a). However the improvement rate increases with BLAST when *CCR* is bigger but is stable for WIEN2K as Table 4.8 shows. As one can tell by Fig. 4.6(d),

the smaller the initial resource pool is the better *AHEFT* outperforms *HEFT*. But once the initial resource is big enough, the improvement rate becomes stable. Another observation is that, the more dynamic the grid environment is, i.e.,the more frequent the new resource is available, the more efficient *AHEFT* can be. Lastly, the improvement rate is not very sensitive to the parameter of β, i.e., the resource heterogeneous factor, and the percentage of resource change, as Fig. 4.6(b) and Fig. 4.6(f) illustrate respectively.

Table 4.7: Improvement rate with various total number of tasks.

Application	200	400	600	800	1000
BLAST	15.9%	18.3%	19.9%	21.9%	23.6%
WIEN2K	2.2%	4.3%	6.0%	7.8%	9.4%

Table 4.8: Improvement rate with various CCRs.

Application	0.1	0.5	1.0	5.0	10.0
BLAST	16.1%	15.5%	14.3%	19.1%	26.1%
WIEN2K	7.3%	7.3%	6.6%	5.3%	6.4%

Overall, the adaptive rescheduling algorithm *AHEFT* outperforms the traditional *HEFT* significantly, and it does even better for workflow applications of high complexity, data intensiveness and parallelism degree in the circumstances of high dynamics and low initial resources, which are exactly the essential characteristics of scientific workflow applications on grids.

4.4 Summary

This chapter analyzes both the benefits and issues of static scheduling strategy for grid workflow applications, and demonstrates how adaptive rescheduling, one feature of the proposed scheduling strategy, can help address the challenge of dynamic resource. In next chapter, we will move on to manage dynamic workload.

CHAPTER 5
SCHEDULING MULTIPLE WORKFLOW APPLICATIONS

In this chapter, the issues of scheduling multiple workflow are discussed in Section 5.1 and a planner-guided dynamic scheduling strategy for multiple workflow applications is proposed in Section 5.2. Section 5.3 presents the experiment design and results which shows that the proposed approach outperforms other dynamic ones such as FIFO and Random significantly.

5.1 Challenges of Scheduling Multiple Workflow Applications

Number of workflows a workflow management system can manage and support is a challenging issue [47]. As the literal survey shows in Section 2.4.3, most static heuristics fail to address this challenge and dynamic ones experience inferior performance.

When a user submits a workflow application, a key question he or she wants to ask is what the *turnaround* time will be, which is measured by the time difference between submission and final completion of the application. In addition, the *makespan* is used to measure the workflow application performance. From a system management perspective, the concern is the overall resource utilization and throughput. While existing dynamic algorithms support dynamic workload allowably consisting of multiple DAGs, their performance is not yet evaluated and comparatively studied with any static counterpart to the best of our knowledge. Given the historic performance evaluation on single DAG scheduling [14, 70, 75, 113], it is not hard to envision that even with multiple DAGs the dynamic algorithms can be optimized if the DAG structure and task execution estimation are taken into account.

As discussed in Section 3.1, the collaboration of workflow *Planner* and *Executor* is the key to manage dynamics. Hence a planner-guided dynamic scheduling algorithm for

multiple workflow applications in Grids is proposed, which inherits both adaptability of dynamic approaches and performance advantages of static ones. With this approach, the workflow *Planner* helps the *Executor* to prioritize tasks globally across multiple DAGs so that the *Executor* is able to assign the task of highest priority to the best resource to achieve better performance. More importantly, it is a practical solution capable of plugging into a real world workflow management system, and it can also be extended to grid environments with applications of mixed varieties.

5.2 A Planner Guided Dynamic Scheduling Algorithm

5.2.1 Planner Guided Scheduling

In the system design, as Figure 3.1 shows, three core components collaborate closely to schedule dynamically and optimize the resource allocation decision: *DAG Planner*, *Job Pool* and *Executor*. The *DAG Planner* assigns each individual task local priority as defined above, manages the task interdependence and submits tasks whenever they are ready to execute into the *Job Pool*, which is an unsorted set containing all tasks from different users waiting to be scheduled. The *Executor* re-prioritizes the tasks in the *Job Pool* before it schedules in the order of task priorities. When a task finishes, the *Executor* notifies the *DAG Planner* which the task belongs to of the completion status. A list of important activities are defined as below:

1. *Job submission*, as defined in Section 3.1.

2. *Job scheduling*. Whenever there are resources available and a task is waiting in the *Job Pool*, the *Executor* will repeatedly do:

   (a) Re-prioritize all tasks currently present in *Job Pool* based on individual task ranks.

   (b) Remove the task with the highest global priority from *Job Pool*;

   (c) Allocate the task to the resource collection which allows earliest finish time.

3. *Job completion notification.* When a task finishes successfully, the *Executor* will notify the corresponding *DAG Planner* of task completion status.

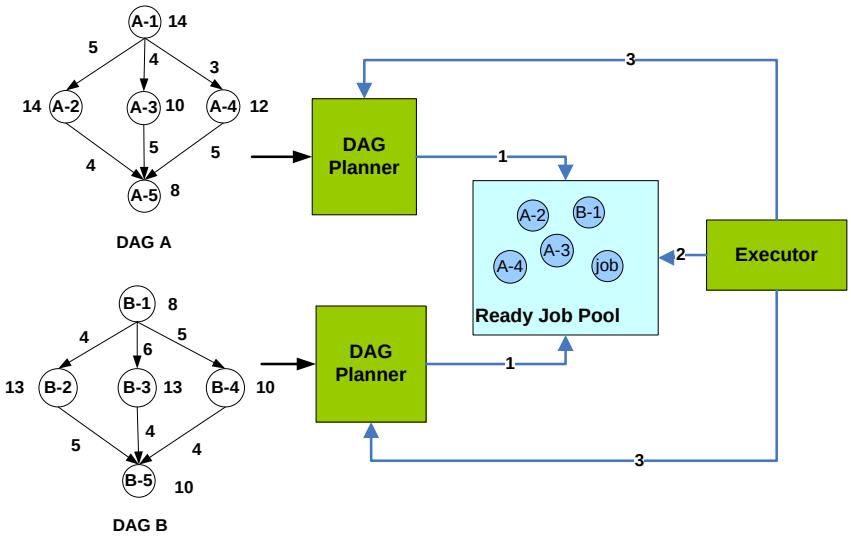


Figure 5.1: An overview of planner-guided dynamic scheduling.

We name this design as *planner-guided dynamic scheduling*. As illustrated in Figure 6.2, each DAG is associated with an instance of *DAG Planner* which ranks individual tasks in the DAG and forwards the ready tasks to the *Job Pool*. If we assume that DAG B arrives in the system right after the task A-1 finishes, the task A-2, A-3, A-4 and B-1 become ready and are submitted into *Job Pool* which may already have tasks from different users. In turn, the *Executor* will re-prioritize all tasks in *Job Pool* before picking the task with the highest global priority. Priority permutation may occur when *Job Pool* makeup changes, for example, a new task from a different user enters into the pool. The next section will detail how to globally prioritize the tasks in the pool.

5.2.2 Task Prioritization

Traditional DAG scheduling algorithms are developed for single DAG domain, directly applying them on multiple DAG scheduling is possible but with great practical limitation. It

is merely equivalent to one of the composition processes discussed in [125]. It creates a composite DAG by making the nodes which do not have any predecessors of all DAGs the immediate successors of a new common entry node, and all the exit nodes of the DAGs immediate predecessors of a new common exit node. A node does not have any predecessor because either itself is an entry node or its predecessors are executing or have finished when composition process occurs. These two extra common nodes have no computation and no communication between them and other nodes. The major difference from [125] is that we consider that DAGs may arrive dynamically in different time. Reusing the examples in Figure 6.2, the composition process will create a composite DAG as illustrated in Figure 5.2. As task A-1 has finished, the common entry node makes itself an immediate predecessor of A-2, A-3 and A-4 from DAG A, B-1 from DAG B and another independent job.

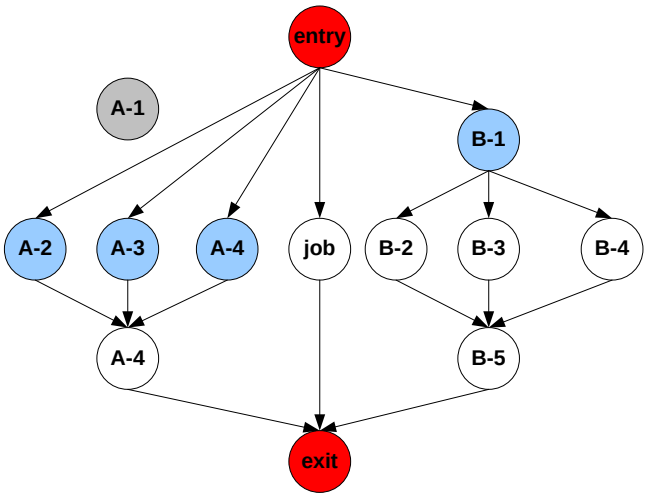


Figure 5.2: An example of DAG composition.

One intuitive approach is to simply apply HEFT on the composite DAG by prioritizing tasks in non_increasing order of rank value. For discussion convenience, we refer to this algorithm as *RANK_HF* in the rest of the paper, which means *the highest rank first*. One can easily recognize that this approach is in favor of:

```
T: a set of tasks in ready job pool
R: a set of free resources
rank: ranking values for all tasks

procedure schedule(T, R) {
  while T ≠ ∅ and R ≠ ∅ do
    boolean multiple = checkMultiple(T);
    if (multiple)
        sort T as an array L so that:
            for any i<j, L[i] ∈ T and L[j] ∈ T , rank(L[i]) ≤ rank(L[j])
    else
        sort T as an array L so that:
            for any i<j, L[i] ∈ T and L[j] ∈ T , rank(L[i]) ≥ rank(L[j])
    endif
    select L[0] ∈ T , where L[0] is the task with the highest priority
    select r ∈ R, where task L[0] has the earliest finish time
                if assigned to resource r
    schedule task L[0] on resource r
    T = T − {L[0]}
    R = R − {r}
  endwhile
}

boolean checkMultiple(T) {
  //This function checks if the tasks in T belong to multiple DAGs
  // return true if tasks belong to multiple DAGs, otherwise return false
}
```

Figure 5.3: The dynamic scheduling algorithm $RANK\_HYBD$.

- The tasks from later arriving DAGs. If the DAGs are of similar complexity, the highest possible rank of a partially executed DAG is very likely smaller than the entry nodes of a newly arriving DAG.

- The tasks that have bigger computation cost. It is obvious that bigger computation cost can help a task to earn higher rank. In an extreme case where all DAGs are a single job type, the priority is actually equivalent to *the longest job first* policy.

However our later experiment shows that such intuitive extension turns out to be the worst performer compared to others in the evaluation. The reason is as follows. As a DAG starts to execute, its rank value of subsequent individual tasks decreases gradually to the lowest point when it reaches to the exit node. If a new DAG or an independent task with big computation cost is submitted in the middle of its execution, the DAG close to completion will not get any resource allocated due to the likely lower global priority until other DAGs are near completion as well. Such policy results in unnecessary longer *turnaround* and *makespan* if the resources are not rich enough, which is validated by the simulation results presented later in this paper.

Based on the observation above and *RANK_HF*'s well known efficiency in scheduling single DAG, we propose a hybrid prioritization algorithm, *RANK_HYBD*, which calculates the global priority based on the rank value of each task in the way as described in Figure 6.3. If there is only one DAG present in the system, *RANK_HYBD* is identical to *RANK_HF*. Otherwise, it prioritizes the tasks in the opposite order. The *Executor* first checks if the tasks in the pool belong to different DAGs. If the tasks come from multiple DAGs, the *Executor* sorts the tasks in a queue (array) which holds the tasks in a non-decreasing order of task ranking value, i.e., the first task has the smallest rank value. If all the tasks belong to the same DAG, the tasks are sorted in opposite order, same as the HEFT algorithm. Then the *Executor* picks the first task in the queue (array) and assigns it to the resource which offers earliest finish time. In an extreme case where all DAGs are actually single tasks, the algorithm is equivalent to the *shortest job first* policy. The later simulation results show

that *RANK_HYBD* improves the average *makespan* and *turnaround* time very impressively while maintaining similar resource utilization and throughput.

We use the two example DAGs in Figure 6.2 to illustrate how algorithm *RANK_HF* and *RANK_HYBD* work. First, a local priority will be assigned to each task by calculating the upward rank values. The task ranking result for each DAG is:

- DAG A: A-1(45); A-2(26); A-3(23); A-4(25) and A-5(8).
- DAG B: B-1(41); B-2(28); B-3(27); B-4(24) and B-5(10).

In this example, we assume that there are two processors, P1 and P2, and DAG B is submitted 6 time units later than DAG A. Figure 5.4 shows the scheduling results.

One may notice that, with the algorithm *RANK_HF*, task A-3 and A-4 are scheduled later as they have lower priorities compared with the tasks from DAG B. As DAG B comes in the midst of execution of DAG A, the tasks on the top level certainly have higher rank values. It supports our observation that *RANK_HF* favors DAGs of later arrival and tasks of more complexity. Basically, the algorithm *RANK_HF* penalizes whichever DAG gets close to completion and results in sub-optimal performance from user's perspective.

Conversely, the *RANK_HYBD* assigns higher priority to the tasks of smaller rank value, which implies that either the task is closer to DAG exit point or the task is less complex, as shown in Figure 5.4(b). When DAG A has started, the remaining tasks, A-2, A-3 and A-4, are the ones on the lower level and therefore have comparatively smaller rank values, compared with entry tasks from DAG B. The *RANK_HYBD* allows the DAG which gets closer to completion higher priority to obtain required resources, at the expense (delaying) of DAGs arriving later or tasks of more complexity though. However, it helps to reduce the majority's *turnaround* and better satisfy users. Finally, it is very fair that when a user submits a new DAG into an already well loaded cluster environment or his DAG request is very complex he would reasonably expect a longer *turnaround* time.

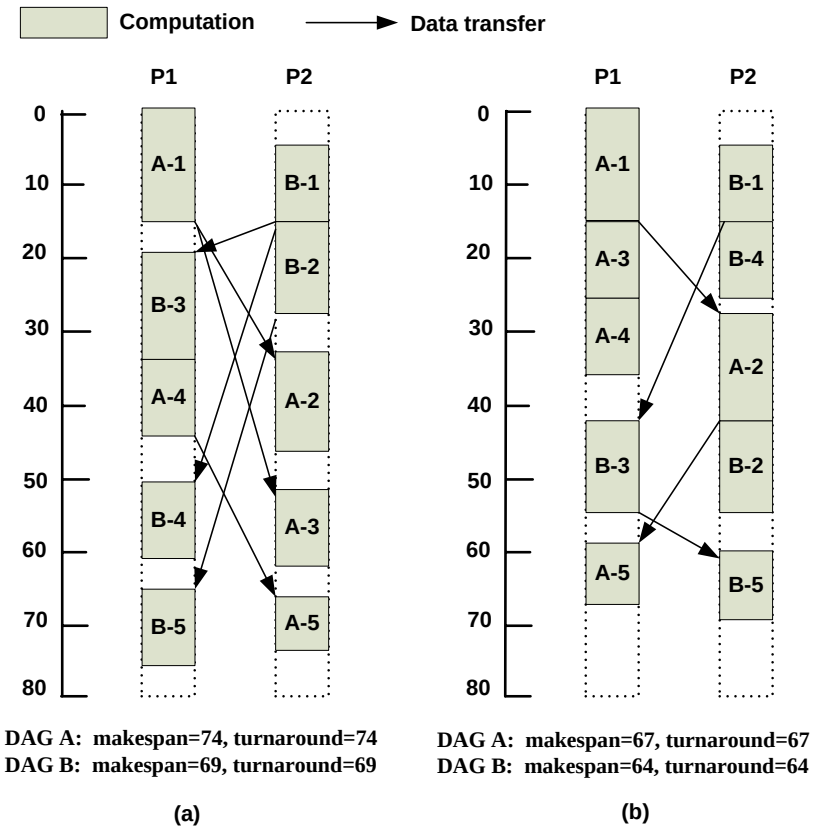


Figure 5.4: Scheduling results: (a) scheduling result for algorithm RANK_HF; (b) scheduling result for algorithm RANK_HYBD.

5.3 Experiment Design and Evaluation Results

In this section, we present the experiment design for evaluating the effectiveness of *RANK_HYBD*. We comparatively evaluate *RANK_HYBD*, *RANDOM*, *FIFO* and *RANK_HF* with published workflow application test bench and analyze the results under an arrange of system parameters.

5.3.1 Algorithms to Evaluate

In order to evaluate the effectiveness of *RANK_HYBD*, we compare it with two practically popular algorithms: *RANDOM* and *FIFO*, along with *RANK_HF*. As a matter of fact, the initial study of *RANK_HF* leads us to design and define *RANK_HYBD* in this paper.

The algorithm details of *RANK_HF* and *RANK_HYBD* are described in Section 5.2. As the name suggests, the *RANDOM* algorithm randomly picks up a task from the *Job Pool* without any priority consideration. With *FIFO*, the *Executor* maintains a queue in the order of task entry time and always chooses the task at the the first place of the queue to schedule. Once a task is selected, the four algorithms adopt the same resource selection process by assigning the task to the free resource which offers earliest finish time.

5.3.2 Workload Simulation

The published test bench [54] for workflow applications is used to evaluate the algorithms. It consists of randomly generated DAGs and is structured according to several DAG graph properties [54]:

- *DAG Size*: the number of nodes in a DAG. As our goal is to evaluate the algorithm performance with complex workload, we use the DAG group with the most tasks only, where each DAG consists of 175 to 249 tasks.

- *Meshing degree*: the extent to which the nodes are connected with each other. It is subdivided into four subcategories: high, medium, low and random.

- *Edge-length*: the average number of nodes located between any two connected nodes. It consists of four subcategories: high, medium, low and random.

- *Node- and Edge-weight.* These two parameters describe the time required for a tasks computation and communication cost. It is related to *CCR*, the communication to computation ratio, but is purposely broken down into Node-high/Edge-high, Node-high/Edge-low, Node-low/Edge-low, Node-low/Edge-high, Node-random/Edge-random subcategories, rather than different CCR values. Obviously, Node-high/Edge-low corresponds to low CCR and Node-low/Edge-high means high CCR.

There are 25 randomly generated DAGs for each combination of subcategories, and they make up totally 2,000 unique test DAGs in our experiment.

The test bench also assumes that each of the available computing nodes, named as target processing elements (TPE) in paper [54], executes just one task at a time and that we have accurate estimates for the computation and communication times of the corresponding DAG scheduling problems [54]. TPE can represent a CPU resource in most contexts of this paper.

Besides the graph properties defined by [54] as above, we add another set of properties to model the dynamic workload:

- *Number of concurrent DAGs.* This is the total number of DAGs concurrently execute in a cluster. We simulate 5, 10, 15, 20 and 25 number of concurrent DAGs respectively in the experiment.

- *Arrival interval.* We are interested in the arrival interval at which the DAGs are submitted into the environment. This is used to mimic the workload dynamics. In the simulation, we assume the the arrival interval follows a Poisson distribution with mean value of 0, 100, 200, 500, 1000, 2000, 3000 and 6000 time units respectively.

With all possible combination of DAG graph properties with dynamic workload characteristics, the experiment involves totally 16,000 test cases based on 2,000 unique DAGs.

Finally, the simulation is developed on top of SimJava [6], an event based simulation framework. It is worth noting that we are targeting a cluster environment in this study, but the proposed scheduling algorithm can be used in a grid of one site with large number of computing nodes, or multiple sites that are connected with a high-speed network, such as TeraGrid [4] and Open Science Grid [5].

5.3.3 Performance Metrics

Since the objective of our algorithm is to improve the workflow application performance, we use the following three metrics to comparatively evaluate all four algorithms:

- *Makespan*: the total execution time for a workflow application from start to finish. It is used to measure the performance of a scheduling algorithm from the perspective of workflow applications.

- *Turnaround time*: the total time between submission and completion of a workflow application, including the real execution time and the waiting time. It measures the performance of a scheduling algorithm from users' perspective.

- *Resource effective utilization*: the ratio of the time for each resource spending on computation to the total time span to finish all DAGs, as defined in Equation 2.4. This metric is used to measure the algorithm efficiency with respect to resource usage from a system perspective.

5.3.4 Simulation Results and Analysis

The simulation results of these four algorithms are compared and analyzed with respect to the evaluation metrics described in previous section against various parameters, including DAG graph characteristics and workload dynamic characteristics of arrival interval and concurrency.

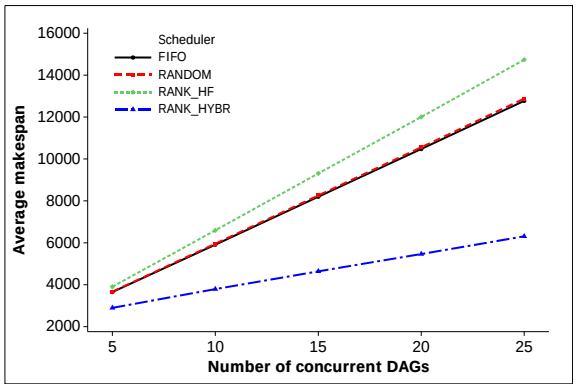


Figure 5.5: Average *makespan* vs. the total number of concurrent DAGs.

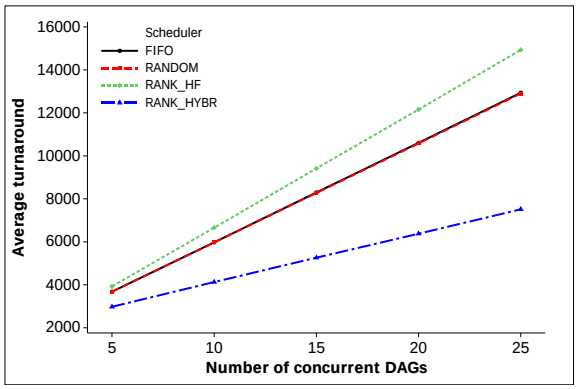


Figure 5.6: Average *turnaround* vs. the total number of concurrent DAGs.

Figure 5.5 and Figure 5.6 show how the algorithms perform with different number of concurrent DAGs. As a result, *RANDOM* and *FIFO* have almost identical performance with respect to average *makespan* and *turnaround*, and they both perform better than the *RANK_HF* algorithm. *RANK_HYBD* always outperforms others and improves even more significantly when the computing environment has more DAGs execute concurrently, with respect to average *makespan*. The average *makespan* improvement rate of *RANK_HYBD* over *FIFO* increases from 20.6% to 50% when total number of concurrent DAGs increases from 5 to 25.

The same observation holds too when the performance is measured by the average *turnaround* time, that *RANK_HYBD* outperforms others better when the system serves more DAGs concurrently, as shown in Figure 5.6. The improvement rate of *RANK_HYBD* over *FIFO* increases from 19% to 41.9% when the total number of concurrent DAGs increases from 5 to 25. With the page limitation and the fact that *turnaround* and *makespan* almost share the identical pattern in the evacuation, in the rest of the paper we will discuss the algorithm evaluation result of both but do not include all *turnaround* metric related figures.

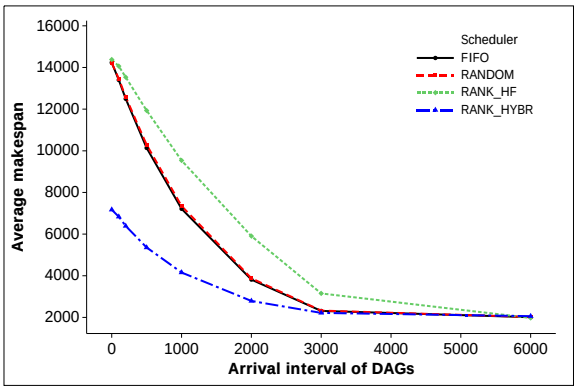


Figure 5.7: Average *makespan* vs. the arrival interval of DAGs.

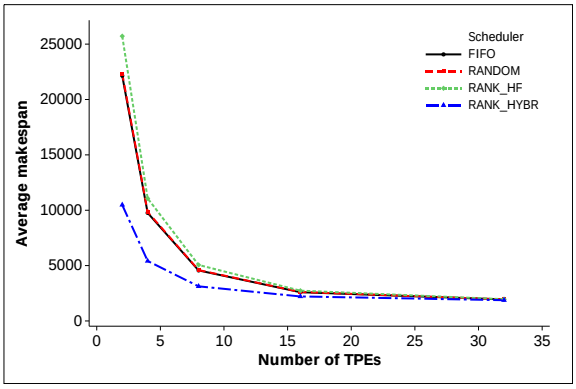


Figure 5.8: Average *makespan* vs. the number of TPEs.

Figure 5.7 helps us to understand how the algorithms respond to workload intensity measured by interval between DAG submission. It can be easily seen that the more intensively DAGs are submitted, i.e., the smaller arrival interval, the better *RANK_HYBD* outperforms other three algorithms in terms of both *makespan* and *turnaround*, as shown in Figure 5.7. When all the DAGs are submitted at the same time, *RANK_HYBD* outperforms *FIFO* by 40% for both average *makespan* and average *turnaround*. Interestingly, once again, *RANDOM* and *FIFO* algorithms have very similar performance. When DAGs arrive at an interval of about 6000 time units, it is almost equivalent to the case that one DAG comes in after another one finishes. In this situation, all of these four algorithms have similar performance. However, *RANK_HF* is the best one and outperforms *RANK_HYBD* by 4% slightly with respect to both average *makespan* and average *turnaround*. But in reality, most high performance computing centers are overloaded.

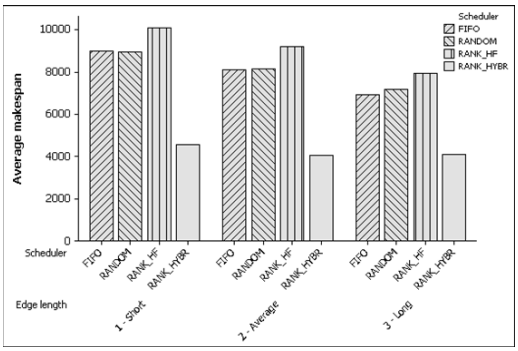
We also investigate how these algorithms perform in terms of resource sufficiency, i.e., the number of TPEs, as shown in Figure 5.8. We once again find out that *RANDOM* and *FIFO* algorithms have similar performance in all scenarios. Figure 5.8 also show that all algorithms do not perform much differently when the cluster environment has sufficient resources, more than 16 TPEs in this experiment. When there are only limited resources available, *RANK_HYBD* is the algorithm of best performance. However, its advantage

diminishes in a fast pace when there are more resources available. As shown in Figure 5.8, its *makespan* improvement rate over *FIFO* drops quickly from 52.6% in the case of two TPEs, to 31.5% in the case of eight TPEs. Same pattern is observed with *turnaround* time, and the improvement rate drops from 43.6% to 27.7% accordingly.

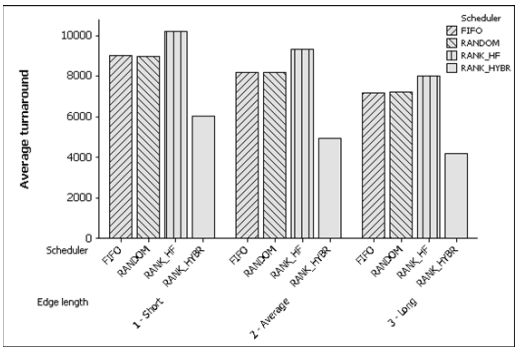
Figure 5.9 shows our further evaluation of *RANK_HYBD* with respect to several DAG graph properties. One can easily observe that *RANK_HYBD* outperforms the other three algorithms in all categories significantly. This evaluation also leads to the discovery that *RANK_HYBD* is less sensitive to DAG graph properties and therefore a relatively fair algorithm in terms of *makespan*. Figure 5.9(a) shows that *RANK_HYBD* results in similar *makespan* for DAGs of different edge lengths. And its performance does not vary much for the DAGs with different meshing degrees either, demonstrated by Figure 5.9(c). For the different communication to computation ratios, as shown in Figure 5.9(e), it has similar performance for DAGs of Node-high/Edge-low Node-low/Edge-high, where the former implies low CCR while the latter indicates high CCR. In terms of average *turnaround* time, all algorithms respond to the each DAG property in a similar way, as Figure 5.9(b), (d) and (f) show.

In addition, we compare *RANK_HYBD*, *FIFO* and *RADOM* with respect to their sensitivity to DAG graph properties measuring in the form of the standard deviation (`Std_Dev`), as shown in Table 5.1. It shows that *RANK_HYBD* is much less sensitive to different DAG properties than *FIFO* and *RANDOM* with respect to average *makespan*. Its sensitivity to *turnaround* is also less than *FIFO* and *RANDOM*, but not significantly. We attribute this to the fact that the *turnaround* time is more related to the system workload, rather than the scheduling algorithm when the system is considerably busy.

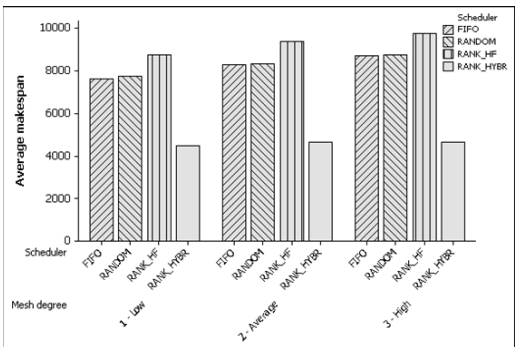
Finally, we study the algorithm performance from the systems' perspective. Figure 5.10 illustrates the empirical Cumulative Distribution Function(CDF) of *resource effective utilization* when the simulation is based on 32 TPEs. The figure shows that all algorithms result in very similar resource utilization percentage. Combined all test results, we conclude



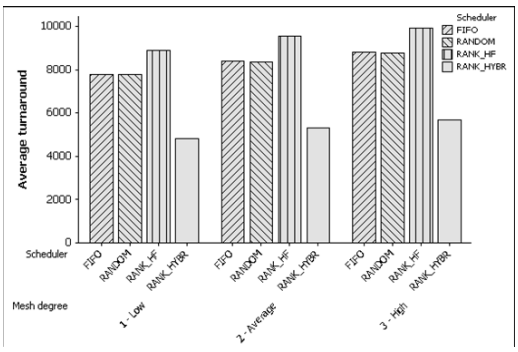
(a) Average makespan vs. edge length



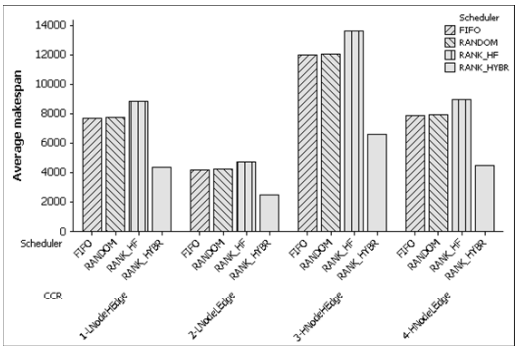
(b) Average turnaround vs. edge length



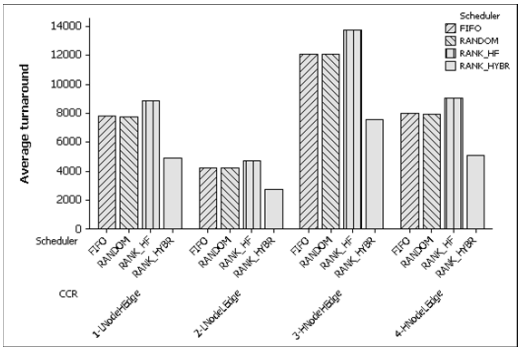
(c) Average makespan vs. mesh degree



(d) Average turnaround vs. mesh degree



(e) Average makespan vs. CCR



(f) Average turnaround vs. CCR

Figure 5.9: Effects of DAG properties on the average *makespan* and *turnaround*.

Table 5.1: Sensitivity to DAG graph properties

DAG property	**Statistic attribute**	***Makespan***		
		RANK_HYBD	**FIFO**	**RANDOM**
Mesh degree	Std_Dev	106.6	544.3	503.9
	Mean	4612.3	8228.0	8302.6
Edge length	Std_Dev	296.0	1032.0	890.6
	Mean	4240.3	8017.8	8121.0
CCR	Std_Dev	1704.1	3181.8	3215.5
	Mean	4489.4	7932.8	8007.4

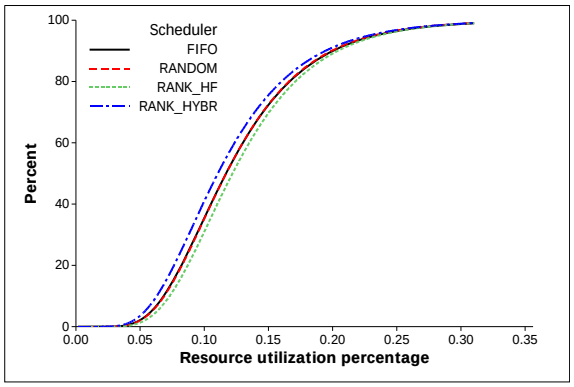


Figure 5.10: CDF for *resource effective utilization* when TPE=32.

that *RANK_HYBD* is the best algorithm, it outperforms *FIFO* and *RANDOM* algorithms by 43.6% and 36.7% with respect to average *makespan* and *turnaround* time respectively.

However, we admit that the conclusion with respect to *resource effective utilization* is not solid as the simulation is performed in a relatively small scale. Moreover, the lack of a proper model of the dynamic workload also makes it difficult to evaluate system performance. For the same reason, we do not further evaluate the throughput metric, as the total number of workflow applications alone does not suffice to properly quantify the real complexity of the work requests. We envision that as more and more workflow applications have been developed and executed on cluster environments, the community will have a better idea of the properties of workflow applications. At that time, it will make more sense to evaluate

the scheduling algorithms from the perspective of systems. We also evaluate the algorithm in terms of fairness, but do not include it due to page limit.

5.4 Summary

This chapter describes how the proposed scheduling strategy addresses the challenge of dynamic workload, i.e., considering that multiple workflow applications arrive dynamically and execute concurrently. The strategy improves dynamic scheduling performance by guiding it with information about workflow structure and job execution time estimation. Meanwhile it is observed that high performance computing platforms, such as OSG [5] and TeraGrid [4], experience high rate of resource failure, which is not well considered in existing heuristics. The next chapter will study how to incorporate resource failure handling into the scheduling strategy.

CHAPTER 6
FAILURE AWARE WORKFLOW SCHEDULING

In this chapter, Section 6.1 discusses why resource failure is important to workflow scheduling. Section 6.2 examines the failure prediction accuracy definition in context of scheduling. Then a failure aware workflow scheduling algorithm is presented in Section 6.3, followed by Section 6.4 which evaluates the performance of the proposed algorithm.

6.1 Resource Failure and Scheduling

With resource failures considered, scheduling workflow applications in a high performance computing system such as cluster and Grid environment is significantly more difficult and unfortunately few existing algorithms tackle this. As a fact, the actual failure rate in production environments is extremely high. Both OSG [5] and TeraGrid [4] report over 30% failures at times [119]. On the other side, the failure tolerance policies applicable to ordinary job scheduling are passively reactive ones and deserve another look as job inter-dependencies in workflows and usually longer computation complicate the failure handling.

Recent years have seen many analysis on published failure traces of large scale cluster systems [45, 68, 87, 101, 124]. These research efforts contribute to better understanding of the failure characteristics, result in more advanced failure prediction models, and help improve system reliability and availability. However, not sufficient attention has been paid to how workflow scheduling can benefit from these accomplishments to reduce the impact of failures on application performance. In particular, we want to answer the following two related questions: One is *what is the right and practical objective of failure prediction in the context of workflow scheduling*? The other is *how does the failure predication accuracy affect*

workflow scheduling? Note that we defer the design of a good failure protection algorithm as our future work.

Failures can have a significant impact on job execution under existing scheduling policies that ignore failures [124]. In an error prone computing environment, failure prediction will improve the scheduling efficiency if it can answer queries with a great success rate such as: "*Will a node fail in next 10 hours if the job is going to execute on this node for 10 hours?*" The job will be assigned to this node only when the answer is "NO" with high confidence. The unexpected failure not only causes rescheduling of the failed job and resource waste on the uncompleted job execution, but also affects the subsequent job assignment which is the key for overall workflow performance. We propose a **F**ai**L**ure **A**ware **W**orkflow scheduling algorithm (**FLAW**) in this paper to schedule workflow applications with resource failure presence.

On the other side, we argue that the conventional definition of failure prediction accuracy does not well reflect how accuracy impacts on scheduling effectiveness. When scheduling a workflow application, the predictor is queried whether a node would fail in a given time window, i.e. the job execution duration. It depends on the capability of the node being assigned to. Typically each individual job has varied computing and communication demands. Moreover, assigned to different nodes the same job may have different execution time decided by node computing capability and data placement. Therefore, a scheduler has to predict potential failure for various nodes for accordingly different time windows. However, the conventional failure prediction result is a set of time periods within which a failure is predicted to happen, different than a scheduler requires. We argue that the conventional approach is not intended for failure aware workflow scheduling and propose two new definitions of failure prediction accuracy: *Application Oblivious Accuracy (AOA)* from a system's perspective and *Application Aware Accuracy (AAA)* from a scheduler's perspective, which we believe better reflect how failure prediction accuracy impacts on scheduling effectiveness.

6.2 Failure Prediction Accuracy

Reliability based failure prediction techniques use various metrics to measure prediction quality, where *precision* and *recall* are popular ones adopted in the literature [66,67,99,100]. The *precision* is the ratio of the number of correctly identified failures to the number of all positive predictions and the *recall* is the ratio of the number of correctly predicted failures to the total number of failures that actually occurred [100]. In other research efforts the accuracy is defined in a statistics context, by measuring how the predicted time between failures is close to actual one [45]. However none of them is intended to be used in job scheduling.

In an error prone high performance computing system, the scheduler requires failure prediction to answer the query before a job is scheduled to the chosen node: *Will this node fail during the job execution?* Intuitively, the quality of failure prediction should be measured by how well those queries can be answered. The scheduler's effectiveness will be adversely impacted if either a failure is not predicted, i.e., the job has to be resubmitted later, or the predicted failure does not actually happen, i.e., a preferred resource may be wasted.

The conventional approach defines a failure prediction is a *true positive* if a true failure occurs within the prediction period Δt_p of the failure prediction [100]. As the conventional definitions originally come from information retrieval theory [100], they depend on the size of Δt_p and do not consider the length of failure down time. Even with the same failure prediction results, the prediction accuracy can vary with the size of prediction period Δt_p.

As illustrated in Figure 6.1, the failure prediction is rated 100% for both *precision* and *recall* given the prediction period Δt_p. But with a smaller predication period $\Delta t'_p$, both *precision* and *recall* change to 50% for the identical prediction.

We want to know if such definition can be used to measure how much failure prediction can impact job scheduling effectiveness and justify what level of accuracy is good enough.

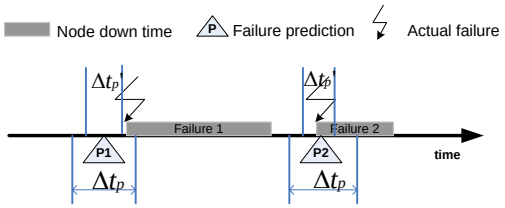


Figure 6.1: An example of actual failure trace and associated failure prediction.

With this in mind, we introduce failure prediction requirements in the context of job scheduling. A failure predictor should be able to predict if a node will fail in the next given time window. The prediction is correct if the node actually goes down in that time window. Before introducing the prediction accuracy, we define three prediction cases as following:

- *True Positive (Hit)*: A failure is predicted and it occurs within the down time of a true failure.
- *False Negative (Fn)*: An actual failure event is not predicted at all.
- *False Positive (Fp)*: A predicted failure does not match any true failure event.

Each failure prediction includes both time and location of predicted failure. By using the same example in Figure 6.1, prediction P1 is a false positive as node is actually alive at the time of P1 predicts. P2 is a hit as it predicts the down time. Failure 1 counts as a false negative as it is not predicted.

$$Accuracy_{AOA} = \frac{Hit}{Hit + Fn + Fp} \tag{6.1}$$

Finally, we define a so called *Application Oblivious Accuracy (AOA)* in Equation 6.1. The failure prediction accuracy in above example is rated as 33.3% accordingly. The definition considers failure downtime, penalizes both false negatives and false positives, and it is measured by failure prediction and actual failures only and therefore more objective.

Furthermore, we observe that the failure prediction with same level of AOA has different impact on job scheduling and the prediction efficiency is application specific, which leads us to define an Application Aware Accuracy (AAA) in Section 6.3.3 later.

6.3 FaiLure Aware Workflow scheduling: FLAW

Inspired by the recent progresses in failure prediction research and increasing popularity of workflow applications, we explore the practical solutions for scheduling a workflow application in an error prone high performance computing environment. In this section, we first discuss the motivation of our research, then describe the solution design and finally illustrate the design by examples.

6.3.1 Motivations

There have been extensive research efforts on workflow scheduling and numerous heuristics are proposed as a result. However, they are yet to address the challenges of scheduling workflow applications in a cluster and grid environment: *dynamic work load* and *dynamic resource availability*.

Following our previous work [122] which tackles the resources dynamics, a DAG scheduling algorithm *RANK_HYBD* [123] is developed to handle dynamic workload in clusters and Grid environments. *RANK_HYBD* is a dynamic scheduling approach which schedules jobs in the order of predefined priority so that the job with higher priority will get preferred resource. Without considering the potential resource failures, *RANK_HYBD* outperforms the widely used (*FIFO*) algorithm significantly in the case of dynamic work load [123].

Furthermore, the design rationale of *RANK_HYBD* provides itself an intrinsic capability to handle resource failures in a proactive way by seamlessly integrating with an online failure predictor. In *RANK_HYBD*, individual jobs are prioritized first to reflect the significance of their respective impact on overall *makespan*, and scheduling decision is made only when a job is ready to execute and resource is available during job execution as predicted. Therefore, failures can be easily handled during scheduling. A workflow typically takes long time to

finish, it is very difficult, if not impossible, for a static scheduling approach to plan for all potential failures in advance. However, resource failures can be much better handled at job level as the job execution time is significantly shorter compared with entire workflow and it is practically easier to predict a failure in shorter period. This assumption is well supported by recent research results [66, 67, 84] which propose failure tolerant scheduling schemes for non-workflow jobs.

On the other hand, the advancement in failure prediction techniques based on analysis of real traces of large scale clusters, is not yet to be leveraged by workflow application schedulers. The profound comprehension of failure patterns and characteristics makes a reasonable accurate failure predictor a practically achievable goal, so for the failure aware workflow scheduler.

6.3.2 Algorithm design

FLAW factors in failure handling by adding an online failure predictor component into the original *RANK_HYBD* design, as Figure 6.2 shows. The proposed system consists of four core components: *DAG Planners*, a *Job Pool*, an *Executor* and an *online failure predictor*. The *Planner* assigns each individual job a local priority as defined in [123], manages the job interdependence and job submission to the *Job Pool*, which is an unsorted collection containing all ready to execute jobs from different users. The *Executor* re-prioritizes the jobs in the *Job Pool* and schedules jobs to the available resources in the order of job priority. When making a scheduling decision, the *Executor* will consult the *Failure Predictor* about whether a resource will keep alive for the entire job execution period if the job is assigned to this resource. If a job is terminated due to unpredicted resource failure, the *Executor* will place the job back into *Job Pool* and the job will be rescheduled. When a job finishes successfully, the *Executor* notifies the *Planner* which the job belongs to of the completion status.

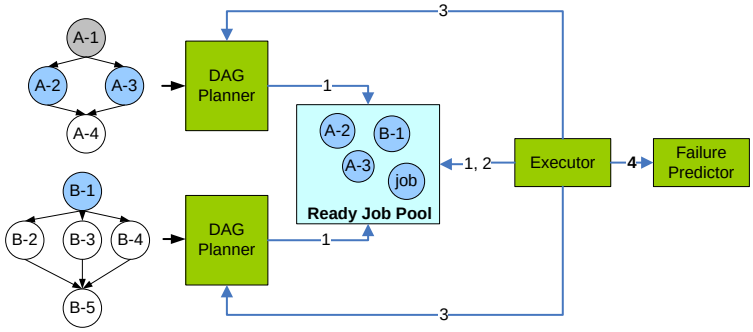


Figure 6.2: An overview of *FLAW* design.

The above collaboration among these core components is achieved by the dynamic event driven design illustrated in Figure 6.2 and explained as follows:

1. *Job submission*, as defined in Section 3.1.

2. *Job scheduling*. Whenever there are resources available and a job is waiting in the *Job Pool*, the *Executor* will repeatedly do:

   (a) Re-prioritize all jobs residing in the *Job Pool* based on individual job ranks in a real time fashion. The jobs in the pool are ready to execute and they may come from different users. The local priority associated with each job will be used to compute global priority. Finally, the jobs in the pool are re-prioritized according to their corresponding global priorities, as defined in [123].

   (b) Remove the job with the highest global priority from *Job Pool* to schedule;

   (c) Schedule the job to the resource which allows the earliest finish time and will not fail during job execution. For the chosen job, the available resources are ordered by the estimated finish time starting from the earliest one. If the resource with higher preference, in terms of estimated finish time, is predicted to fail during the job execution, the next resource will be attempted. One may notice that the job execution time varies with resource and so does the failure prediction time

window. If none of the resources can keep alive during the period of the chosen job execution, this job will remain in the *Job Pool* and next job will be picked out for scheduling. Otherwise, the job is scheduled and will run on the assigned resource. Figure 6.3 describes this scheduling algorithm in more details.

3. *Job completion notification.* When a job finishes successfully, the *Executor* will notify the corresponding *DAG Planner* of job completion status.

4. *Failure Prediction.* The *Failure Predictor* will answer queries coming from the *Executor*: Will the resource X fail in next Y time units? Y is the estimated job execution time if the job is scheduled on resource X. The answer "YES" or "NO" drives the *Executor* make completely different scheduling decisions and therefore impose potentially great impact on the effectiveness of scheduler and overall application performance as well.

As each design comes with predefined objectives, the design of *FLAW* is to:

- *Reduce the loss time.* Accurate failure prediction will help the scheduler avoid placing jobs on a resource to fail in the middle of job execution. The abnormally terminated execution contributes to system resource waste, i.e., loss time caused by failures, including time spending on both unfinished data transmission and computation.

- *Reduce the number of job rescheduling.* Checkpointing/restarting is a simple and popular fault tolerance technique, but it incurs considerable overhead of lower system utilization and work loss. Indeed, the checkpointing can be more detrimental than the failures themselves and it should be more intelligent [85]. As the number of nodes grows, failure rate will increase and the current checkpointing techniques has to cope with speedup and data compression issues otherwise total system cost will increase significantly in the future [102]. Therefore checkpointing is not utilized in our system design. Instead, a failed job will be rescheduled later and start over. The number

of *job rescheduling* can measure how a scheduler benefit from failure prediction from another angle. We envision that this metric will be of great interest to domain experts who are using cluster and Grid environment.

- *Reduce the makespan.* The *makespan* is the overall performance indicator for workflow applications and the effectiveness measure of a workflow scheduler.

```
T: a set of jobs in ready job pool
R: a set of free resources
rank: ranking values for all jobs

procedure schedule (T, R) {
  sort T as an array L so that:
        for any i<j, L[i] ∈ T and L[j] ∈ T , rank(L[i]) ≥ rank(L[j])
  FOR i=1 TO size of L
    calculate the earliest finish time of L[i] on each resource r ∈ T
      if L[i] is assigned to r, and sort R as an array of N in increasing
      order of the estimate earliest finish time
    FOR j = 1 TO size of N
      predict if the N[j] will fail when job L[i] runs on N[j]
      IF YES
        INCREMENT j
      ELSE
        schedule job L[i] on resource N[j]
        T = T − {L[i]}
        R = R − {N[ j]}
      END IF
    INCREMENT i
  END FOR
}
```

Figure 6.3: The scheduling algorithm in *FLAW*.

*6.3.3 Application Aware Accuracy (*AAA*)*

The key success factor to the *FLAW* design is the accuracy of failure prediction, which is measured by how effectively the *Failure Predictor* can answer the query: "Will the resource X fail in next Y time units?" The effectiveness of failure prediction can be quantified by the ratio of correct answers to total queries in context of job scheduling. It is worth noting that *how to predict* failures accurately and effectively is not the goal of this paper. Instead, we intend to find out what is the practical requirement for failure prediction from the prospective of scheduling and how a scheduler can leverage failure prediction in an error prone environment.

Different than the AOA defined earlier, the above ratio is application and prediction timing specific. For example, assuming that the present time is at time unit 0 , a node will go down between 100 and 120 time units and a job can be completed on this node by 140 time units if starting from now. It is further assumed that the *Failure Predictor* forecasts a failure will occur at time unit of 130, which is actually a false positive. In this case, the *Failure Predictor* can still give a correct answer to the query "if the node will be down in next 140 time units?" by telling "Yes".

We referred to this ratio as *Application Aware Accuracy* (AAA) and use it to measure the failure prediction effectiveness. Even though a higher AOA helps improve AAA, however, the AAA highly depends on the application behaviors and how and when the query is made. In an extreme example, if a resource is highly error prone and none of the failures on this resource is predicted, the AAA can still be very high if this resource is never a preferred one and no query is made about it. This sounds strange but can be very true in workflow scheduling. For instance, in a resource rich environment a node with very low capability can not produce completive earliest finish time for any job and is hardly considered in scheduling. And for a data intensive workflow application, in order to reduce cost on data movement the scheduler may narrow the resource choices to certain nodes which have executed many jobs and retain the data for next dependant jobs.

6.3.4 An Example of Failure Aware Scheduling

In this section, we illustrate the *FLAW* design by using examples of a workflow application and a failure trace, as shown in Figure6.4. It is assumed that the sample DAG will run in a environment consisted of three nodes. The nodes encounter some failures as defined in the box of the figure.

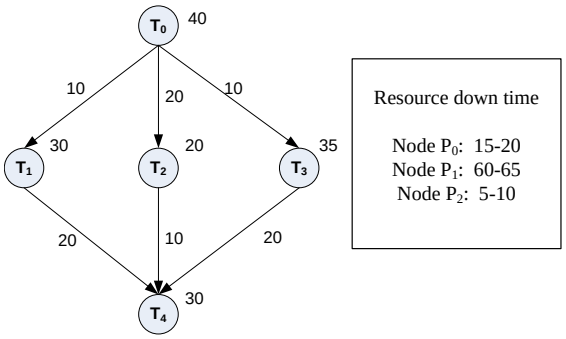


Figure 6.4: Example of a DAG and failure trace.

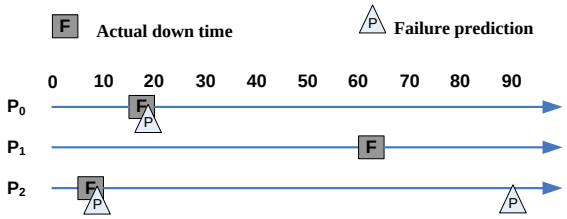


Figure 6.5: Failure prediction with 50% of *AOA*.

We further assume that a *Failure Predictor* makes the following failure prediction: node P_0 fails at 18, node P_2 fails at 8 and 90 respectively, as shown by Figure 6.5. This prediction achieves *AOA* accuracy of 50%, which includes 2 hits, 1 false positive and 1 false negative.

Figure 6.6 gives the scheduling result by the *RANK_HYBD* algorithm without failure prediction and the *FLAW* algorithm with presence of failures defined. It shows that the *FLAW* finishes the sample application with 150 time unit *makespan*, 20 time unit loss time and 1 job rescheduling. *FLAW* outperforms *RANK_HYBD* in all areas which completes with *makespan* of 155, loss time of 30 and 4 job rescheduling.

A detailed trace which records how each scheduling decision is made is illustrated in Figure 6.7 and Figure 6.8 for *RANK_HYBD* and *FLAW* respectively. *FLAW* finishes with 90% of *AAA* accuracy, as there are totally 10 queries and only one false negative prediction is made.

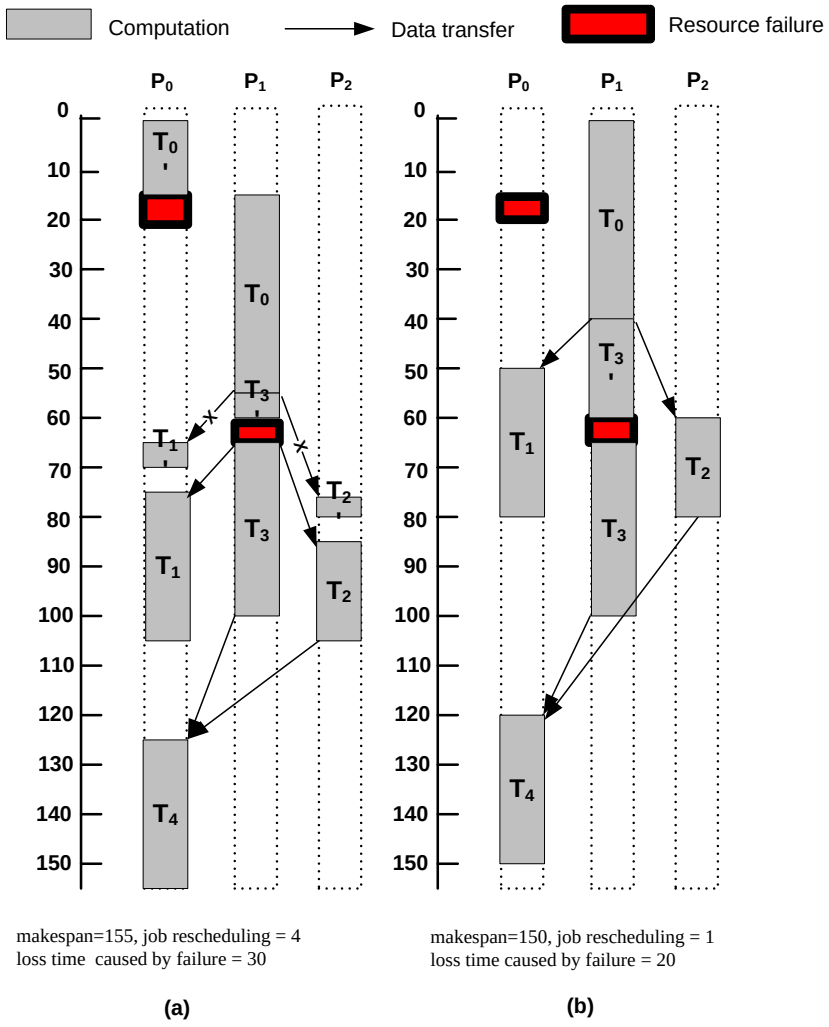


Figure 6.6: Scheduling results: (a) *RANK_HYBD* without failure prediction; (b) *FLAW* with failure prediction of 50% of AOA.

Time/Event	Job Pool	Prioritized queue	Resource Pool	Failure Prediction	Scheduling Decision
0 DAG A arrive	$\{T_0\}$	$\{T_0\}$	$\{P_0, P_1, P_2\}$	N/A	**(T_0, P_0)**
15 P_0 fails (T_0 is terminated)	$\{T_0\}$	$\{T_0\}$	$\{P_1, P_2\}$	N/A	**(T_0, P_1)**
20 P_0 recovers	{ }	{}	$\{P_0, P_2\}$	N/A	
55 T_0 done	$\{T_1, T_2, T_3\}$	$\{T_3, T_1, T_2\}$	$\{P_0, P_1, P_2\}$	N/A	**(T_3, P_1) (T_1, P_0) (T_2, P_2)**
60 P_1 fails (Data transfer for T_1 and T_2 are terminated)	$\{T_1, T_2, T_3\}$	$\{T_3, T_1, T_2\}$	$\{P_2\}$	N/A	**NA (The data required by T_1, T_2 and T_3 resides on P_1)**
65 P_1 recovers	$\{T_1, T_2, T_3\}$	$\{T_3, T_1, T_2\}$	$\{P_0, P_1, P_2\}$	N/A	**(T_3, P_1) (T_1, P_0) (T_2, P_2)**
……	…..	…..	….	…..	**……**
105 T_2 done	$\{T_4\}$	$\{T_4\}$	$\{P_0, P_1, P_2\}$		**(T_4, P_0)**
155 T_4 done	**{}**	**{}**	**$\{P_0, P_1, P_2\}$**		**NA**

Figure 6.7: *RANK_HYBD* scheduling trace.

6.4 Performance Evaluation and Analysis

To verify the design of *FLAW* and study how failure prediction accuracy affects the scheduling effectiveness, we present the simulation design and result analysis in this section.

6.4.1 Workload Simulation

The published test bench [54] for workflow applications is used in the simulation. The test bench consists of randomly generated DAGs and is structured according to the following DAG graph properties:

- *DAG Size*: the total number of jobs in a DAG. As our goal is to evaluate the algorithm performance with intensive workloads, we only use the DAG group with the most jobs, i.e. the DAG consists of 175 to 249 jobs.

- *Meshing degree*: the extent to which the nodes are connected with each other.

Time/Event	Job Pool	Prioritized queue	Resource Pool	Failure Prediction	Scheduling Decision
0 DAG A arrive	$\{T_0\}$	$\{T_0\}$	$\{P_0, P_1, P_2\}$	P_0 will fail at 15	**(T_0, P_1)**
15 P_0 fails	{}	{}	$\{P_2\}$	N/A	**NA**
20 P_0 recovers	{ }	{}	$\{P_0, P_2\}$	N/A	**NA**
40 T_0 done	$\{T_1, T_2, T_3\}$	$\{T_3, T_1, T_2\}$	$\{P_0, P_1, P_2\}$	None of node will fail	**(T_3, P_1) (T_1, P_0) (T_2, P_2)**
60 P_1 fails (T_3 is terminated)	$\{T_3\}$	$\{T_3\}$	{ }	N/A	**NA**
65 P_1 recovers	$\{T_3\}$	$\{T_3,\}$	$\{P_1\}$	N/A	**(T_3, P_1)**
……	…..	…..	….	…..	**……**
100 T_3 done	$\{T_4\}$	$\{T_4\}$	$\{P_0, P_1, P_{02}\}$		**(T_4, P_0)**
150 T_4 done	**{}**	**{}**	**$\{P_0, P_1, P_2\}$**		**NA**

Figure 6.8: *FLAW* scheduling trace.

- *Edge-length*: the distance between the connected nodes, i.e., the average number of nodes located between the sender and receiver.

- *Node- and Edge-weight*. These two parameters describe the time required for a jobs computation and communication cost and are related to *CCR*, the communication to computation ratio.

As our simulation focuses on how to handle failures in scheduling, the DAGs we choose for this simulation are those being random on all properties of meshing degree, edge-length and node-weigh and edge-weigh. According to the test bench description [54], both *Meshing degree* and *Edge-length* are distributed uniformly with the lower bound of 1% and the upper one of 100%. In order to utilize the real failure trace with granularity of minutes, we treat one time unit in DAG as 5 minutes. And we have multiple workflow application execute concurrently in the simulations to mimic the dynamic workload in reality.

The test bench [54] provides DAGs for test targeting environment of different scale measured by the total number of Target Processing Elements (TPE). A TPE can safely represent a node in a cluster system. We choose the DAGs designed for 32 TPEs in this simulation as this is a popular cluster scale in practice.

6.4.2 Failure Traces and Prediction Accuracy

Studies in [45, 68, 87, 101, 124] recognize the temporal and spatial correlation of failures in large scale cluster systems. In order to mimic these failure characteristics in the simulations, we choose to use the failure traces published by Los Alamos National Laboratory [71].

In the simulation we extract 10 two-month failure traces from the real failure trace [71] by randomly picking up 32 nodes out of 49 nodes in the Cluster 2 and randomly choose two-month period for these 32 nodes between calendar year 2001 and 2002.

For each real failure trace, we randomly generate an associated prediction trace which is planed with random mixture of hits, false-positives and false-negatives to simulate different levels of AOA, i.e., 50%, 60%, 70%, 80% and 90% of AOA respectively. Five prediction traces are generated for each real failure trace at a defined AOA level. The simulation uses 10 actual failure traces and 250 generated failure prediction traces. Finally, the level of AOA is simulated by that the *Predictor* looks up prediction traces to answer the query.

6.4.3 Performance Metrics

The evaluation is designed to study what is the right objective of failure prediction and how failure prediction affects the scheduling effectiveness, and the following metrics are measured against different levels of AOA:

- *Makespan*, which is the total execution time for a workflow application from start to finish. It is used to measure the performance of a scheduling algorithm from the perspective of workflow applications.

- *The loss time*, which is defined as the total time of partial execution including both data transmission and computation. It measures the system resource waste caused by resource failures.

- *The number of rescheduling jobs*, which is defined as the total number of job rescheduling which is caused by resource failures. If a node fails in the middle of job execution, the job is terminated and placed back to job pool for rescheduling.

- *Corresponding AAA*, which measures the effectiveness of failure prediction.

6.4.4 Simulation Results and Analysis

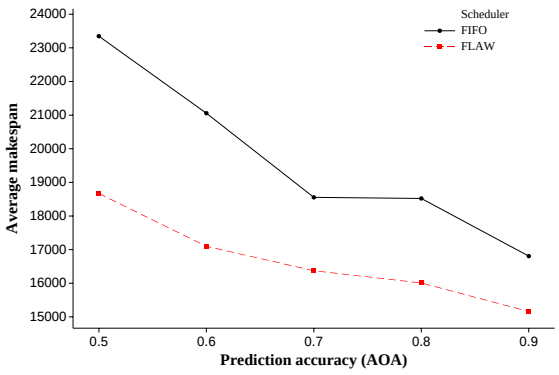


Figure 6.9: *FIFO* vs *FLAW* with 10 concurrent DAGs.

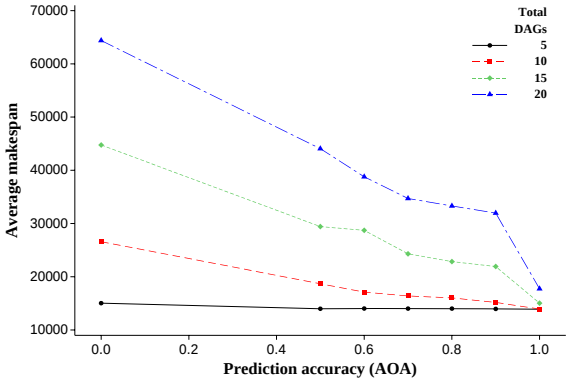


Figure 6.10: Average *makespan* vs prediction accuracy (*AOA*).

Our previous work [123] demonstrates that *RANK_HYBD* outperforms *FIFO* without resource failure presence, the simulation result in Figure 6.9 further proves that *RANK_HYBD* based failure aware scheduler, i.e., *FLAW*, outperforms *FIFO* based one in terms of *makespan* when 10 concurrent DAGs are running in the system. As our interest is studying the impact of failure prediction, we do not further evaluate *FIFO*.

Most of the analysis below is to evaluate performance metrics against different levels of *AOA*. The simulation also includes two extreme situations: 1) *AOA* is 0, which means the scheduler is failure blind and does not predict failure at all (i.e., the basic *RANK_HYBD*); 2) *AOA* is 1.0, which means the scheduler knows exactly failure happens by looking up the actual failure trace.

It can be easily seen that with AOA increasing, workflow applications perform better in terms of *makespan*, as shown in Figure 6.10. As the work load intensity increases measured by the number of concurrent DAGs, the performance improvement is even bigger.

Figure 6.11 shows that, on average, higher level of AOA helps *FLAW* reduces the *loss time* considerably. Similarly, the number of *rescheduling jobs* is improved with more accurate failure prediction as shown in Figure 6.12. Figure 6.13 reports the impact of predication accuracy on *makespan* of different workloads. We can see that the advantage of failure predication increases as the workload increases.

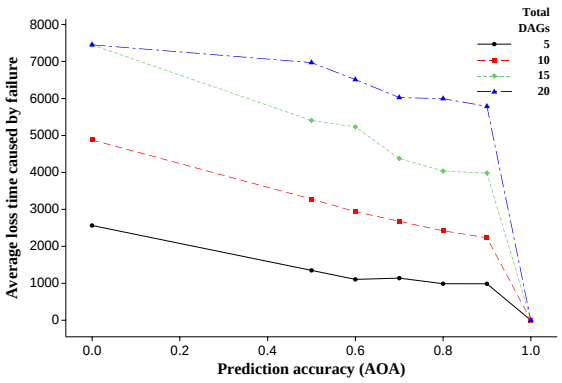


Figure 6.11: Average *loss time* vs. prediction accuracy (AOA).

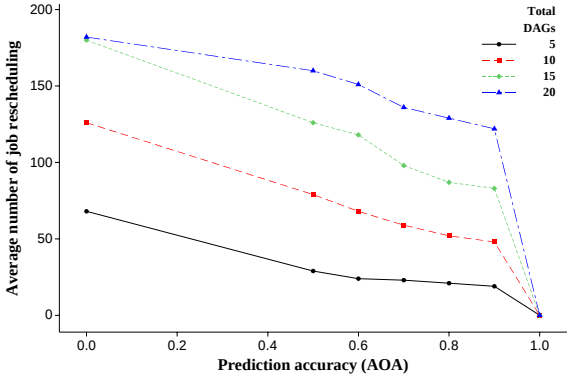


Figure 6.12: Number of *job rescheduling* vs. prediction accuracy (AOA).

We also study whether false positives or false negative has more significant impact on scheduler effectiveness. In order to do that, for each one of 10 real failure traces, additional 10 prediction traces are generated with full spectrum of possible mixtures of Fp and Fn. The simulation is performed against total 200 generated traces with AOA levels of 50% and 60%. The Figure 6.14 does not tell any correlation of average *makespan* and the percentage of $F_p/(F_p + F_n)$.

Finally we try to understand what is the right and practically achievable objective of failure prediction accuracy. Figure 6.15 shows that the effectiveness measured by AAA is about 96% when AOA is as low as 50%, which indicates that a high AAA can be achieved with moderate AOA. As work load gets more intensive, a failure blind scheduler (i.e., AOA=0.0)

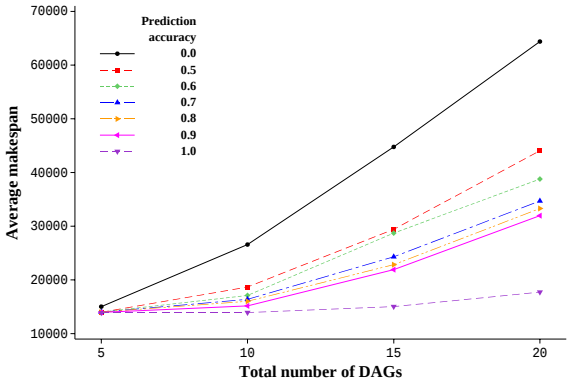


Figure 6.13: Average *makespan* vs. the total number of DAGs.

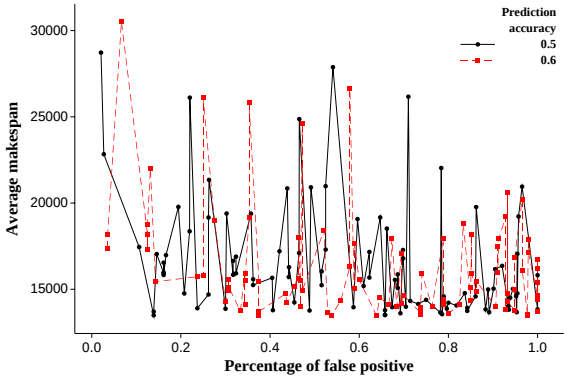


Figure 6.14: Average *makespan* vs. the percentage of false positive.

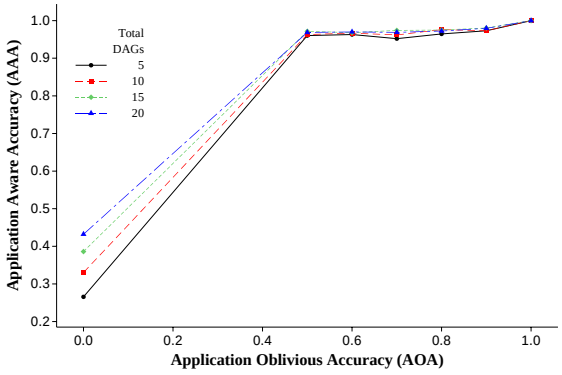


Figure 6.15: The comparison between AOA and AAA.

can accomplish closer to 50% of AAA, and the AAA rate is more stable and increases steadily as AOA improves and $FLAW$ performs well even with trivial AOAs.

6.5 Summary

In this chapter, we incorporate proactive failure handling into the scheduling strategy, and propose two new definitions of failure prediction accuracy in the context of workflow scheduling. It is worth noting that the simulations so far is conducted on a cluster of 32 nodes. However, the experiment results apply to more popular grid environments as well, such as cluster of clusters. In next chapter, we evaluate how the proposed scheduling strategy supports scheduling workflow applications in a cluster of clusters environment.

CHAPTER 7
WORKFLOW SCHEDULING ON MULTICLUSTERS

This chapter discusses how to augment the algorithm proposed earlier in response to a new form of Grids, a cluster of clusters or multicluster environment. Section 7.1 introduces the challenges of scheduling workflows in a multicluster environment and a scheduling algorithm is proposed in Section 7.2. Section 7.3 describes the experiment design to evaluate the effectiveness of the proposed algorithm. Finally, evaluation results and analysis is presented in Section 7.4.

7.1 Challenges of Scheduling Workflows on Multiclusters

At an early stage of the cluster and grid technology, PC cluster was the majority form as Figure 2.2 shows. One of the initial purposes of grid computing technology is to utilize the idle PCs and otherwise wasted computation time. For example, the Condor was developed to [21]: First, it makes available resources more efficient by putting idle machines to work; Second, it expands the resources available to users, by functioning well in an environment of distributed ownership.

As the demand for computation power keeps growing and the infrastructure matures rapidly, some new trends in Grid computing are recognized as below:

1. Multicluster environment is becoming more popular. Initially, clusters were developed in form of interconnected workstations on a local area network. Nowadays, with considerable progresses made to support security and resource access across domains, particularly introduction of the Globus Toolkit Version 4 [38], clusters are connected with a high speed backbone to build more powerful high performance and high throughput computing platforms, where each cluster manages its workload independently in its

own domain and in its own right. In reality, most clusters are equipped with batch job schedulers, such as PBS Professional [88], MOAB(formerly Maui) [78], Platform LSF [72] etc. For example, Condor-G combines the inter-domain resource management protocols of the Globus Toolkit [48], intra-domain resources and job management methods of Condor to allow the user to harness multi-domain resources as if they all belong to one personal domain [22], as depicted in Figure 7.1. The term domain, site and cluster are used interchangeably in the remaining discussions.

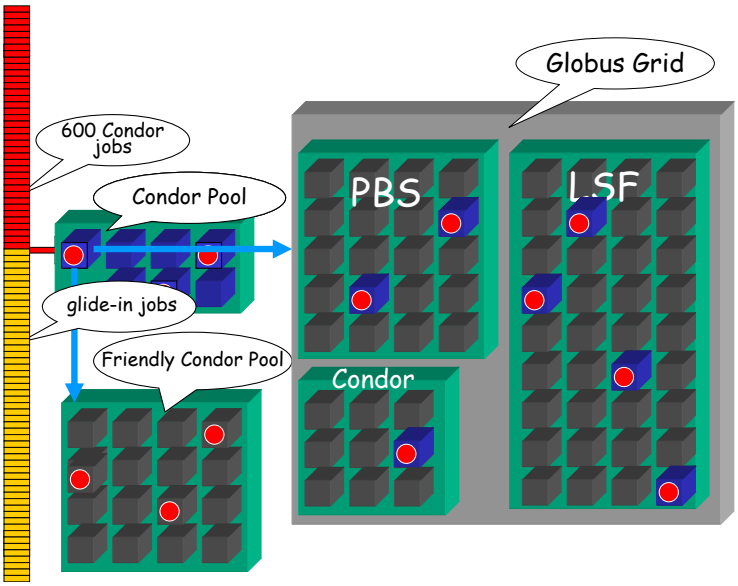


Figure 7.1: Condor-G: interface with other scheduling system across clusters [44].

2. With advancement in parallel programming technology, task-parallel execution has been shown successful on both homogeneous and heterogenous parallel systems for many applications, which provides a suitable degree of multiprocessor task parallelism [93]. The task in this research context is the one which can be assigned to a number of available processors, referred to as M-task. Recent years have seen more research on scheduling mixed-parallel application on Grids [8, 79, 109].

The potential that a task can be executed with different degrees of multi-processor parallelism brings both new opportunities and challenges to workflow scheduling. On one hand, the ability to utilize more processors in execution helps to improve workflow performance. On the other hand, it is a new challenge in terms of how workflow scheduling can leverage this:

1. A typical scientific workflow application involves non-trivial data stage-in and stage-out activities. When data movement crosses clusters, it is possibly not worth simply pursuing more resource provision from other clusters if the data communication cost overweighs the reduction of execution time. Most previous heuristics assume a shared data storage in a cluster, which can be still true but the extra data movement crossing clusters has to be considered. The scheduler has to make tradeoff decisions when facing the choices of either executing the related tasks on the same cluster with fewer processors or dispatching the task to a different cluster of richer resource at extra cost of data transportation.

2. In a multicluster environment, each cluster has its own workload management system and there is always dynamic (background) workload locally. A task submitted to a cluster will be subject to its local workload management policy. A task may have to wait in the queue for considerable time period before getting executed. However, the previous research always assumes the task will be executed immediately without waiting in the queue once it is scheduled. On the other hand, it has been a challenge to predict how long a job will wait in the queue. Fortunately, there is a breakthrough on the queue wait time prediction practice [16, 80, 81], which has been implemented on TeraGrid [4] as Batch Queue Prediction Web service (QBETS Service) querying all queues of multiple clusters.

3. Both of the technology advancements listed above together make the necessity for scheduling workflow applications in a multicluster environment. The potential benefit

of utilizing multiclusters can not be realized without the ability of predicting the job queue wait time, data movement cost and execution time save.

One may note that previous heuristics always schedule tasks to a single TPE, or a single processor. While traditional dynamic scheduling algorithms do not consider the job queue wait time, even worse, the fundamental assumptions made by static algorithms are not valid at all in this situation. For example, static heuristics unrealistically assume that resources are always available with full capacity. The misconceptions and unrealistic assumptions are analyzed in the work [52]. However, we envision that our proposed collaborative workflow scheduling strategy, i.e., a hybrid of static and dynamic strategy, can be augmented to handle multi cluster situation and further improve the workflow performance.

The strategy proposed in this dissertation considers task priorities, tunable resource requirements, queue wait time prediction and presence of background workload. Its goal is to minimize *makespan* and improve *resource utilization efficiency*.

7.2 A Strategy for Scheduling Workflows on Multiclusters

7.2.1 System Design

In order to manage scheduling workflow applications on a multicluster environment, we propose a system design which is augmented from the earlier version, as shown by Figure 7.2.

Similar to the system design described in Chapter 3, the scheme of planner guided dynamic scheduling is employed and supported by the following core components:

Meta Scheduler manages job admission, enforces the inter-task dependency, prioritizes tasks, constructs appropriate resource requirements and selects suitable resource to achieve minimal overall makespan. It is further broken into three major components:

Planner ranks each individual task of a workflow and ensures that tasks submitted to the *Job Pool* are ready to execute, as outlined in Section 3.1. The rank value reflects how importantly a task can impact the overall workflow makespan. A

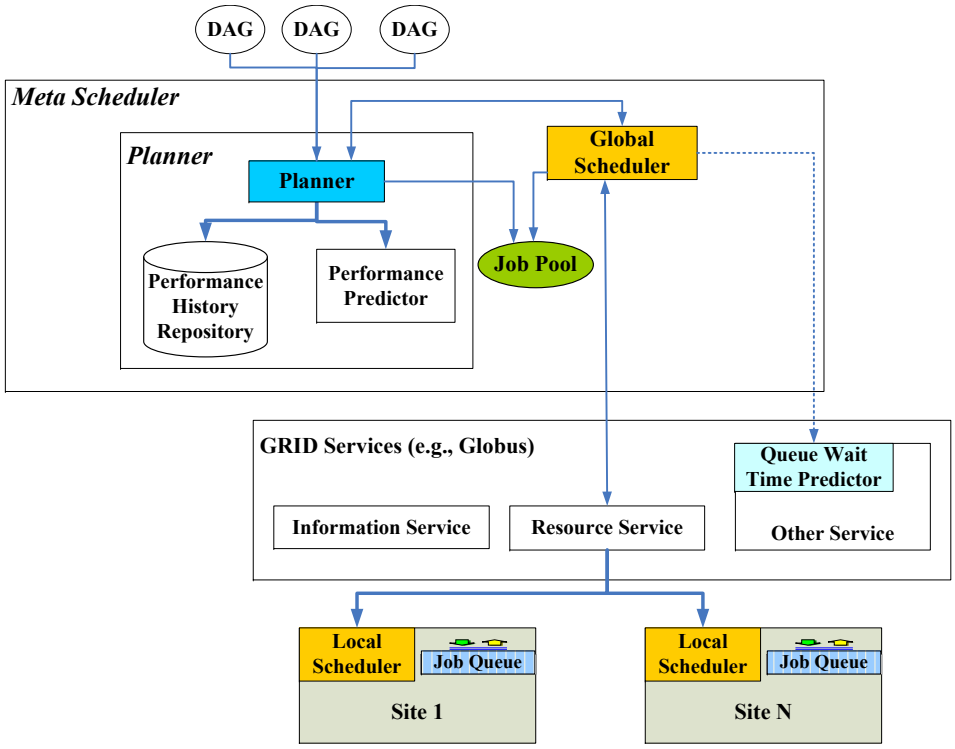


Figure 7.2: The conceptual system design of DAG scheduling on multi sites.

task with higher rank value has higher priority and will be scheduled earlier. The ranking algorithm employed here is defined in Section 3.2. In addition, the *Planner* is responsible for specifying the resource requirements and estimating the execution performance accordingly, which is detailed later in Section 7.2.2. As a result, each task may be submitted with a rank value and multiple appropriate resource requirements. The requirements may be different from each other on the number of resources to request.

Job Pool contains all jobs ready to execute in an unsorted manner, as defined in Section 3.1.

Global Scheduler , as the most vital component in this system, picks up the task with the highest priority and selects the most suitable resource. In a multicluster environment, the *Global Scheduler* does not have direct control over resources.

Instead, it only dispatches tasks to the independent site/cluster with appropriate resource requirements accordingly. Therefore, it is very crucial to schedule tasks based on priority and be able to predict the queue wait time on targeted clusters.

Grid Service includes softwares and components supporting security, information infrastructure, resource management, data management, communication, fault detection, and portability. As one of the popular grid service softwares, the Globus Toolkit [48] core services, interfaces and protocols allow users to access remote resources as if they were located within their own machine room while simultaneously preserving local control over who can use resources and when. One important service here is the "Queue Wait Time Predictor", which can relatively accurately predict how long a job has to wait in the queue for resource requirement fulfillment. A real world implementation of such service is the QBET Network Weather Service [91] deployed on TeraGrid [4], which can predict the upper bound of wait time given the resource requirement.

Local Scheduler refers to the independent workload management system on each individual cluser/site. It receives tasks dispatched by the *Global Scheduler* and treats these tasks as any other local jobs submitted by the users directly to this cluster. No difference from local jobs, the workflow tasks are subject to the local scheduling policy, which in most cases is first-come-first-served(FCFS) queuing. In this context, the *Global Scheduler* cab be considered as a special system user of the cluster.

7.2.2 Resource Specification and Performance Model

With recent years witnessing Grids of rich resources in growth, the single processor based heuristics hardly leverage the resource abundance. However, when an application can access more resources, it is still difficult for a user to describe the resource requirements. [56] presents an empirical model that allows a DAG based workflow application to generate appropriate resource specifications, including the number of resource, the range of clock rates

among the resources and network connectivity. This is referred to as *Tunable Requirement* in this dissertation.

While [56] proposes a "Size Prediction Model" that predicts the best number of resources to use in resource requirement specifications sent to a scheduler, it admits that predicting the "best size" is challenging. How to generate resource specifications is also one of the identified challenges in the NSF workshop [28].

In order to accommodate the tunable requirement specifications, we need redefine the performance model by extending the traditional one defined in Section 2.1. To simplify the performance modeling, we assume that each processor has same clock rate [56].

Furthermore, we reference the Amdahls' law discussion in [53] which assumes efforts that devote r resources will result in sequential performance $\sqrt{r}$. In other words, if $w_{i,j}$ is the estimated time to complete task n_i on processor r_j, and task n_i will be scheduled to n processors with same capacity as r_j, the execution time of task n_i will be $w_{i,j}/\sqrt{n}$. Thus, the performance can get double with four time number of processor assigned. It is worth noting that, [53] tried other similar functions (for example, $\sqrt[1.5]{r}$) but found no important changes to their results. This simplified performance model will be used in the forthcoming discussion and experiment design.

7.2.3 Queue Wait Time Prediction

Queue wait time prediction has drawn tremendous research efforts [30, 31, 105, 115] in last decade. But it did not show significant impact on scheduling strategy design until the recent deployment of QBETS [80, 81] on more than a dozen super computing sites, offering two types of on-line queue delay predictions for individual jobs:

1. Given the job characteristics, QBETS can predict a statistical upper bound on how long the job is likely to spend waiting in the queue prior to execution;

2. Given the job characteristics and a start deadline, QBETS can calculate the probability that the job begins execution by the deadline.

As QBETS service provides API and web service for external integration, we can safely assume an on-line queue time prediction service readily available and the *Global Scheduler* can inquiry the service of queue wait time for any resource requirement. The query result is the input for *Global Scheduler* to decide where to dispatch the job and with what resource requirement, more specifically, the number of processors to request.

7.2.4 Scheduling Algorithm

With the performance model and queue wait time prediction component in place, a detailed scheduling algorithm is defined in Figure 7.3.

As Figure 7.3 shows, the *Global Scheduler* picks up the task with the highest priority, i,e., the highest rank value, to schedule first. Different than traditional heuristics which assume a task is only to execute on a single processor, the *Planner* in this system potentially defines multiple appropriate resource requirement specifications for any individual task. It is understandable, and actually some parallel programs specify number of processors as command line arguments. In this dissertation we assume that, if a task has multiple appropriate resource requirements, the requirements differentiate each other solely on the number of processors to request.

The basic concept of the proposed algorithm is to schedule tasks on the "best" resource collection, which is measured by the smallest earliest finish time (EFT). One can easily deduce that the makespan of a workflow is the maximum EFT value of all exit tasks. In this context, a resource collection is qualified one if it meets all the resource requirements first, such as operating system type, storage space, network bandwidth etc.. The best one minimizes the EFT with all potential costs and benefits being taken into account, including the performance adjustment with request of different number of processors, consequently resulted different queue wait time and possible data movement cost, described as below:

$$EFT(t) = clock() + exec(t, q, p) + comm(t, S(q)) + QBETS(S(q), q, p) \qquad (7.1)$$

For each appropriate resource requirement, the *Global Scheduler* checks all qualified resource collections to determine the EFT for each by estimating:

- *Execution time* $exec(t, q, p)$. This is estimated based on the performance model defined in Section 7.2.2. We assume the estimated execution time for task t on single processor in queue q is $W_{t,q}$ and $exec(t, q, p) = \frac{W_{t,q}}{\sqrt{q}}$.

- *Communication cost* $comm(t, q)$. It is very common that a local cluster has a centralized share data storage. Therefore, if a task and all its parent tasks execute on the same cluster *S(q)*, the communication cost will be zero, otherwise there is a cost for data movement between clusters. The *Global Scheduler* has to balance resource availability and possible data transfer cost.

- *Queue wait time prediction QBETS(S(q), q, p).* This service returns predicted queue wait time for a task if it is scheduled to queue q on cluster S with request for p processors. It is worth noting that a typical cluster S may contain multiple queues. As a common sense, a request for larger number of processors may result longer wait time in the queue. So trade off has to be made between gaining on performance and wait in the queue.

The proposed algorithm is a natural augmentation from the planner guided dynamic scheduling [122, 123] by adding queue wait time awareness. However, how to schedule multiple workflows is not discussed here for two reasons: First, as matter of fact, each cluster may receive workflows locally and schedule them within its domain, it is equivalent to scheduling multiple workflows. Second, here we focus on how to schedule workflows with dynamic background workload across multiple clusters and the algorithm presented here shares same philosophy with the one defined in [123].

7.3 Experiment Design

This section presents the experiment design for evaluating the effectiveness of the proposed algorithm. We first evaluate the algorithm with queue wait time awareness verse without, and then study how tunable resource requirement affects the performance with prediction ability.

7.3.1 Assumptions

To evaluate how the tunable requirement and queue wait time prediction can improve scheduling effectiveness, we conduct simulation based experiments with DAS-2 cluster workload traces [65] and published workflow test bench [54]. The simulation is developed with a well known workflow simulation framework GridSim [107].

As an explorative study, the following assumptions are made in the simulation:

1. It is assumed that all tasks can execute on any processor. In other words, each processor meets the basic resource requirements of individual task.

2. Each task may request various number of processors as part of the resource requirements. Once scheduled, a task can not change its resource requirement over time.

3. Each individual task is scheduled to a single cluster. No task will execute across multiple clusters.

4. The local scheduler implements FCFS with simple backfilling policy. Jobs are queued in the order of job arrival time. When there are free resources available, the first job in the queue will be evaluated and scheduled if there is sufficient number of processors available. Otherwise, the job will stay at its original location in the queue until sufficient resources become available. However, the next job can backfill if its request can be accommodated. So on and so forth for the remaining jobs in the queue.

5. If a task's parent(s) execute on different cluster(s), it is required to transfer the parent task's data output to the cluster where the task is scheduled to. The data transfer starts right after the task arrives. It is assumed that the Grid service transfers file(s) from the remote cluster(s) as requested.

6. We assume the queue wait time prediction capability in the system design. The experiment predicts the queue wait time for a job by simulating the execution sequence of the jobs in the queue. With regard to prediction accuracy, we are not able to run a statistic comparison against QBETS due to unavailability of actual data of QBETS. However, the prediction accuracy is insignificant in the context of workflow scheduling and not a focus here.

7.3.2 Workload Simulation

The real workload traces from the DAS-2 is used in the experiment. The DAS-2 supercomputer consists of five clusters located at five Dutch universities and is primarily used for computing and scientific research. The largest cluster (Vrije Universiteit) contains 72 nodes and the other four clusters have 32 nodes each. Every node contains two 1GHz Pentium III processors, 1GB RAM and 20GB local storage. The clusters are interconnected by the Dutch university internet backbone and the nodes within a local cluster are connected by high speed Myrinet as well as Fast Ethernet LANS. All clusters use openPBS [86] as local batch system (one and only one queue is configured for each cluster). Maui [78](FCFS with backfilling) is used as the local scheduler. Jobs that require multi-clusters can be submitted using toolkits such as Globus [48]. DAS-2 runs RedHat Linux as the operating system. A comprehensive analysis of the workload on DAS-2 is available in [65].

One may notice that most assumptions made in Section 7.3.1 are rightfully valid with the DAS-2 environment. An overview of the DAS-2 system and workload traces is provided in Table 7.1.

Table 7.1: DAS-2 clusters and workload traces (A'dam - Amsterdam) [65].

Cluster	Location	#CPUs	Period	#Job Entries
fs0	Vrije Univ. A'dam	144	01-12/2003	219618
fs1	Leiden Univ.	64	01-12/2003	39356
fs2	Univ. of A'dam	64	01-12/2003	65382
fs3	Delft Univ. of Tech.	64	01-12/2003	66112
fs4	Utrecht Univ.	64	02-12/2003	32953

From the original DAS-2 workload traces, we randomly extract four 24 hour period (from 14:00 to 14:00 next day) traces with various workload intensity and add a special zero workload trace for the simulation. Workload trace detail is provided in Table 7.2.

Table 7.2: Simulation trace details.

Trace#	#Jobs(fs0)	#Jobs(fs1)	#Jobs(fs2)	#Jobs(fs3)	#Jobs(fs4)	Total #Jobs
0	0	0	0	0	0	0
1	3823	857	1170	277	431	6558
2	2166	850	819	261	801	4897
3	4191	656	1632	483	219	7181
4	4701	18	17	3505	1911	10152

Each job entry in the workload trace contains the following information: the job number, the job arrival time, the job start time, the job run time, the job completion status, the number of processor requested and allocated, the cluster assigned to, and so on. All jobs in the traces are exactly simulated according to these information except for the job start time. The jobs from traces may experience longer wait time in the queue as there are extra jobs from workflows to compete resources.

7.3.3 Performance Metrics

The following performance metrics are used in the simulation to evaluate the effectiveness of the proposed workflow scheduling strategy:

- *Makespan*, which is the total execution time for a workflow application from start to finish. It is used to measure the performance of a scheduling algorithm from the perspective of workflow applications. The primary design objective of a scheduling algorithm is to minimize the *makespan*.

- *Data transfer time*, which is a measure of total time of data transfer cross clusters. Data movement occurs when the child tasks are scheduled to different clusters which either accommodate the requests or have better resource provision. But it consumes network resources and should be minimized.

- *Queue wait time*, which is a measure of total time workflow tasks wait in the queue. The *queue wait time* for a task is the time difference between when it arrives in the queue and when it gets executed. A task waits in the queue because: it observes the FCFS policy to wait its turn to get executed and/or it waits for the required data being transferred to the cluster where it is scheduled to.

- *Resource Effective Utilization*, which is a measure of the fraction of used processor cycles with respect to its best possible usage during the workflow execution, as defined in Equation 2.4. The *resource effective utilization* also takes the background workload into account but is only measured for the period of workflow execution in this simulation.

7.3.4 Workflow Simulation

The published test bench [54] for workflow applications is used to evaluate the proposed algorithm as well. It consists of randomly generated DAGs and is structured according to several DAG graph properties [54]:

- *DAG Size*: the number of nodes in a DAG.

- *Meshing degree*: the extent to which the nodes are connected with each other.

- *Edge-length*: the average number of nodes located between any two connected nodes.

- *Node- and Edge-weight.* These two parameters describe the time required for a jobs computation and communication cost. It is related to the communication to computation ratio (CCR).

The test bench also assumes that each of the available computing nodes executes just one job at a time and that we have accurate estimates for the computation and communication times of the corresponding DAG scheduling problems [54].

7.3.5 Scenarios to Evaluate

As the previous work in Chapter 4, Chapter 5 and Chapter 6 already demonstrate that the planner guided dynamic scheduling well outperforms the popular FIFO and Random algorithms, we do not compare against them again. In this experiment, we simulate different scheduling scenarios defined in Table 7.3 with focus on understanding how the proposed algorithm benefits from the queue wait time awareness in a multicluster environment and the impact of tunable resource requirements.

Table 7.3: Scheduling scenarios to evaluate.

Scenario Name	#processors to request	Queue wait time predictable
16_no	16	No
32_no	32	No
48_no	48	No
16_yes	16	Yes
32_yes	32	Yes
48_yes	48	Yes
Tun_yes	16, 32 or 48	Yes

For the first six scenarios of 16_no, 32_no, 48_no, 16_yes, 32_yes and 48_yes, each task has a single resource requirement which specifies the number of processors to request. As the name suggests, 16_no always requests 16 processors and is blind to queue wait time. In

other words, in the algorithm defined in Figure 7.3, the $QBETS(S(q), q, p)$ function always returns value of 0 in scenarios of 16_no, 32_no and 48_no. In another scenario Tun_yes, each task has multiple requirements, i.e., requesting for 16, 32 and 48 processors respectively. To accommodate the tunable requirements, the algorithm should be able to predict the queue wait time. It is worth noting that jobs observe the local FCFS scheduling policy in all clusters in these scenarios.

7.4 Evaluation Results and Analysis

Now we are in a position to depict the evaluation results. The published test bench [54] for workflow applications is used to evaluate the algorithm. The DAG group with most number of tasks (175 to 249 tasks) in chosen in the simulation. The time unit used in the following discussion is a second.

7.4.1 Experiment Results

We first simulate all scenarios with the five workload traces described in Table 7.2 and fifty DAGs with all characteristics (Meshing degree, Edge-length, Node- and Edge-weight) being random. Figure 7.4 and Table 7.4 illustrate workflow performance with regard to the average *makespan* in different scenarios. With workload trace 0, a special case of zero background workload, the algorithm performs almost identically with all scenarios. With the real workload in presence, measured by average *makespan* the algorithm performs much better with queue wait time awareness than without. Overall, with the same number of processors requested, the algorithm improves average *makespan* 3 to 10 times when prediction is enabled.

In terms of total *queue wait time*, as shown in Figure 7.5, tasks spend way more time waiting in queues when the algorithm is not able to predict, regardless of how many processors tasks get allocated. This contributes in a considerable portion to the long *makespan*. It is easily understandable that the queue wait time awareness helps reduce job wait time significantly.

Table 7.4: Average *makespan* in various scenarios.

Trace	**Scenario**						
	16_no	16_yes	32_no	32_yes	48_no	48_yes	Tun_yes
0	36960	38591	31911	34459	32751	34891	35091
1	201213	49694	214323	56239	222057	71116	62553
2	564665	50632	539747	47565	587511	52182	48308
3	172945	60972	170767	64049	177499	79821	67370
4	225438	41455	217923	37449	232031	39132	38498

However, good performance is achieved at the cost of extra time spent on data movement across clusters, as shown by Figure 7.6. Data movement is more active in the scenarios with prediction enabled. This is the trade off between performance gain and extra data movement.

To better understand how the algorithm performs in different scenarios with various workload traces, Table 7.5 provides an overview of performance measurements on average *makespan*, total *queue wait time*, total *data transfer time* and total number of data transfer requests. It is observed that:

1. There is an evidently strong correlation between average *makespan* and total queue wait time. The less queue wait time is, the less average *makespan*. This leads to an obvious conclusion that reducing the queue wait time is crucial to minimize *makespan*, a key performance index for workflow applications. It is worth noting that we are not able to break down the *makespan* to the degree at which we can tell exactly how much queue wait time contributes to overall *makespan* for two reasons: First, the *makespan* measures overall workflow execution time but queue wait time is tracked at individual task level. Second, a task waits in a queue for various reasons as specified in Section 7.3.3.

2. Average *makespan* is reduced at the cost of data movement. Except for Trace 0, the scenario with the smallest *makespan* always has very high total data transfer time

Table 7.5: Performance metric measurement overview

Workload trace	Scenario	Average makespan	Total queue wait time	Total data trans. time	Total data trans. req.
Trace 0	*16_no*	36,960	3,722,845	562,475	3,789
	32_no	31,911	3,201,416	577,399	3,884
	48_no	32,751	3,324,266	626,558	4,216
	16_yes	38,591	3,876,476	671,343	4,515
	32_yes	34,459	3,398,933	706,630	4,751
	48_yes	34,891	3,527,649	835,131	5,624
	Tun_yes	35,091	3,512,064	818,749	5,508
Trace 1	*16_no*	201,213	23,709,857	601,802	4,052
	32_no	214,323	26,555,817	605,951	4,079
	48_no	222,058	33,704,178	600,329	4,048
	16_yes	49,694	4,938,243	840,182	5,628
	32_yes	56,240	5,561,998	742,825	4,975
	48_yes	71,116	7,463,092	605,096	4,046
	Tun_yes	62,553	6,112,858	708,145	4,738
Trace 2	*16_no*	564,666	44,543,689	645,187	4,345
	32_no	539,747	42,591,350	598,828	4,030
	48_no	587,511	57,810,894	618,654	4,165
	16_yes	50,633	4,999,785	854,209	5,729
	32_yes	47,566	4,692,228	924,222	6,202
	48_yes	52,183	5,057,724	887,498	5,954
	Tun_yes	48,309	4,795,350	902,275	6,047
Trace 3	*16_no*	172,945	22,545,631	597,888	4,027
	32_no	170,767	23,672,800	613,256	4,128
	48_no	177,499	27,299,651	594,290	4,003
	16_yes	60,973	5,974,432	821,762	5,503
	32_yes	64,049	6,790,100	853,395	5,707
	48_yes	79,821	8,351,560	796,916	5,328
	Tun_yes	67,370	7,109,935	807,695	5,405
Trace 4	*16_no*	225,438	32,829,228	595,784	4,008
	32_no	217,923	31,233,502	598,896	4,033
	48_no	232,031	34,573,599	621,371	4,184
	16_yes	41,456	4,148,387	790,517	5,310
	32_yes	37,450	3,745,788	856,970	5,762
	48_yes	39,132	3,861,621	901,959	6,051
	Tun_yes	38,498	3,841,241	879,100	5,906

and the number of data transfer requests. The data movement happens if the total reduction of execution time and queue wait time can offset the extra time spent on data movement. However, we have to admit that this dissertation only considers performance from the perspective of execution time and it does not consider the consumption of other network resources such as network bandwidth.

3. Queue wait time prediction is critical when background dynamic workload is considered. For trace 0, which is a special case of zero background workload, the algorithm performs almost identically with or without prediction. Particularly, the scenario of 32_no has the best performance. This does not cast any doubt on the necessity of queue wait time prediction. Contradictorily, it shows that existing heuristics, which do not consider any of resource competition and dynamic workload, are not justified realistically.

4. Dynamic scheduling is inherently nearsighted. The dynamic scheduling decision is to minimize the EFT of each individual task. With this guidance, a task can be simply dispatched to a different cluster with data movement required to reduce the EFT for this task only, but the decision may hurt the overall performance of entire workflow. This observation is supported by: no single scenario outperforms others in all cases; high volume data movement helps reduce *makespan* but does not warrant that the scenario with the highest total time of data transfer always has the least *makespan*. Even so, the dynamic scheduling is the only viable choice in real world when both dynamic resource and dynamic workload are considered. The key to improve its effectiveness is the collaboration between the *Planner* and the *Executor*.

With the queue wait time prediction, the average *resource effective utilization* improves as well for all clusters. As shown in Figure 7.7, the average *resource effective utilization* of cluster fs0 is better with queue wait time awareness than without. The observation holds true for other clusters as well. One may note that the average *resource effective utilization* is

extremely low in all scenarios with trace 0 simply because there is no background workload at all. As trace 1 and 3 present intensive workload on cluster fs0, it is relatively high for all scenarios.

7.4.2 Cumulative Distribution Analysis

We further study how these performance metrics distribute in the experiment on 50 DAGs with all attributes being random. For this purposes, the cumulative distribution function (CDF) figures of the *makespan* and the *queue wait time* metric are created for trace 0 and 4, as shown in Figure 7.8. Trace 0 is a special case with no other workload involved, and trace 4 is a representative one for real workload traces with intensive background workload.

It is noticed that all CDF curves fit well with linear form, indicating the validity of evaluating overall performance by average values of corresponding performance metrics in Section 7.4.1. For trace 4, which is a typical representative of all traces except trace 0, 99% of *makespan* is about 59607, 48902, 52974 and 52901 for scenario 16_yes, 32_yes, 48_yes and Tun_yes respectively, 99% of the *queue waiting time* is about 6514318, 5618301, 6012874 and 6051709 respectively. Both are significantly reduced comparing with the scenarios without queue wait time awareness.

Secondly, Figure 7.8 once again shows that, with intensive dynamic workload, the queue wait time prediction helps reduce the *queue wait time* drastically which results in very big improvement on *makespan* at relatively very low cost of extra *data transfer time*.

These CDF figures also help better understand how tunable resource requirement impacts scheduling. Even though the Tun_yes is not the best performer in all cases with respect to average *makespan*, it actually leads in steepness of cumulative distribution curves, evidently in Figure 7.8(a) and (c). If we take out lower 10% and upper 10% of the test cases to minimize possible abnormality in simulation, the Tun_yes is actually the steepest scenario with respect to *makespan* and *queue wait time* in both traces, as shown in Figure 7.8(a),(b),(c) and (d). As all prediction enabled scenarios perform closely if measured

by average *makespan*, the steepness actually places Tun_yes as a leading performer in terms of the *makespan* and *queue wait time*.

Furthermore, we try to understand how CCR impacts the scheduling effectiveness. Fig. 7.9 shows that in terms of the average *makespan* the proposed algorithm performs noticeably best with scenario Tun_yes when CCR is high (HNodeLEdge), i.e. computation intensive, and worst when CCR is low (LNodeHEdge), compared with other scenarios. This indicates that when a workflow has high CCR, the *Planner* intends to request more processors. The observation also suggests that the resource requirement specification should consider the workflow characteristics intelligently, particularly the CCR ratio.

7.4.3 Discussion of Tunable Requirements

It is interestingly observed that Tun_yes is not the best performer as we thought. It actually has similar performance as 16_yes, 32_yes and 48_yes. We further study how Tun_yes requests the number of processors when it has options of 16, 32 or 48 processors. Table 7.6 shows that the algorithm requests 48 processors most of time. We believe this partially attributes to nearsightedness of dynamic scheduling, which always makes local decisions. On the other hand, we admit that the combination of 16, 32 and 48 is only an educated guess. Ideally, the resource requirements of a workflow are determined specifically by its application characteristics. However, the experiment here is performed with DAGs randomly generated and we use one set of resource options for all cases. In reality, scientific workflows are well studied and the recently emerging studies, such as [56], will help generate "best" resource specifications.

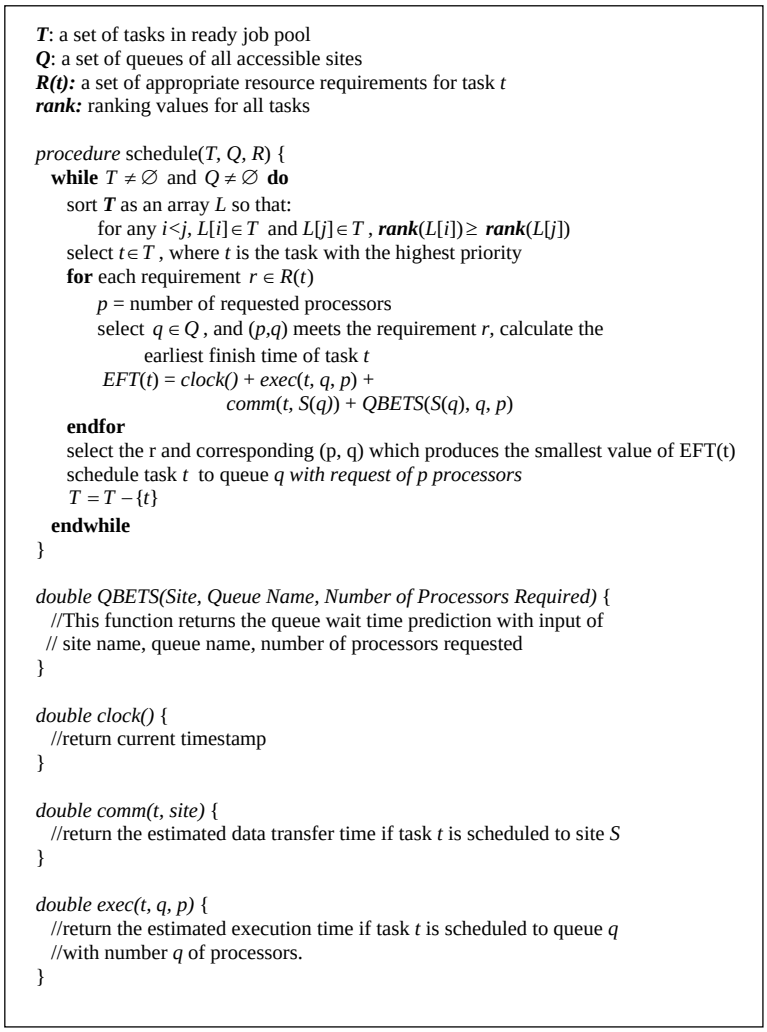
Table 7.6: Distribution of processor number request.

Trace	Percentage of requests for 16 processors	Percentage of requests for 32 processors	Percentage of requests for 48 processors
Trace 1	8.2%	19.5%	72.2%
Trace 2	1.3%	5.6%	93.1%
Trace 3	8.6%	18.5%	72.9%
Trace 4	1.0%	1.9%	97.1%

The experiment results conclude that queue wait time awareness is crucial for scheduling workflows in a multicluster environment and the proposed algorithm performs extremely well with prediction awareness. It also suggests that future progress on how to better generate workflow resource requirements can further improve the proposed algorithm.

7.5 Summary

As high performance computing environments are growing into the form of a cluster of clusters in recent years, this chapter attempts to augment the proposed scheduling strategy in response to the new trend. It introduces the concept of *Global Scheduler*, which is still guided by the *Planner* but also has to collaborate with each individual independent local cluster scheduler. In next chapter, we will present a Java based prototype to demonstrate the scheduling strategy can be integrated with Condor [21].



T: a set of tasks in ready job pool
Q: a set of queues of all accessible sites
R(t): a set of appropriate resource requirements for task *t*
rank: ranking values for all tasks

procedure schedule(*T, Q, R*) {
  while $T \neq \varnothing$ and $Q \neq \varnothing$ **do**
    sort ***T*** as an array *L* so that:
      for any $i<j$, $L[i] \in T$ and $L[j] \in T$, ***rank***$(L[i]) \geq$ ***rank***$(L[j])$
    select $t \in T$, where *t* is the task with the highest priority
    for each requirement $r \in R(t)$
      p = number of requested processors
      select $q \in Q$, and (*p,q*) meets the requirement *r*, calculate the
          earliest finish time of task *t*
       EFT(*t*) = *clock()* + *exec*(*t, q, p*) +
                  comm(*t, S*(*q*)) + *QBETS*(*S*(*q*), *q, p*)
    endfor
    select the r and corresponding (p, q) which produces the smallest value of EFT(t)
    schedule task *t* to queue *q with request of p processors*
    $T = T - \{t\}$
  endwhile
}

double QBETS(Site, Queue Name, Number of Processors Required) {
  //This function returns the queue wait time prediction with input of
 // site name, queue name, number of processors requested
}

double clock() {
  //return current timestamp
}

double comm(t, site) {
  //return the estimated data transfer time if task *t* is scheduled to site *S*
}

double exec(t, q, p) {
  //return the estimated execution time if task *t* is scheduled to queue *q*
  //with number *q* of processors.
}

Figure 7.3: Algorithm of DAG scheduling on a cluster of clusters.

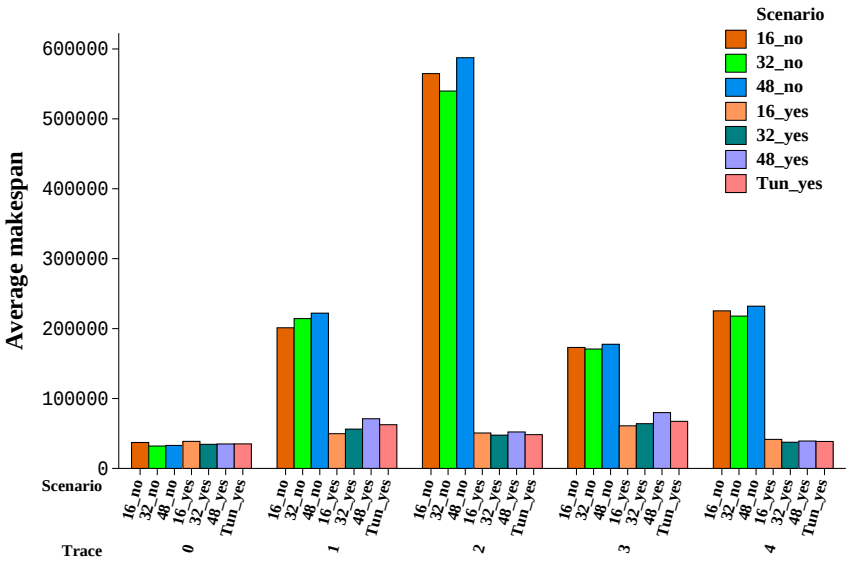


Figure 7.4: Average *makespan* in various scenarios.

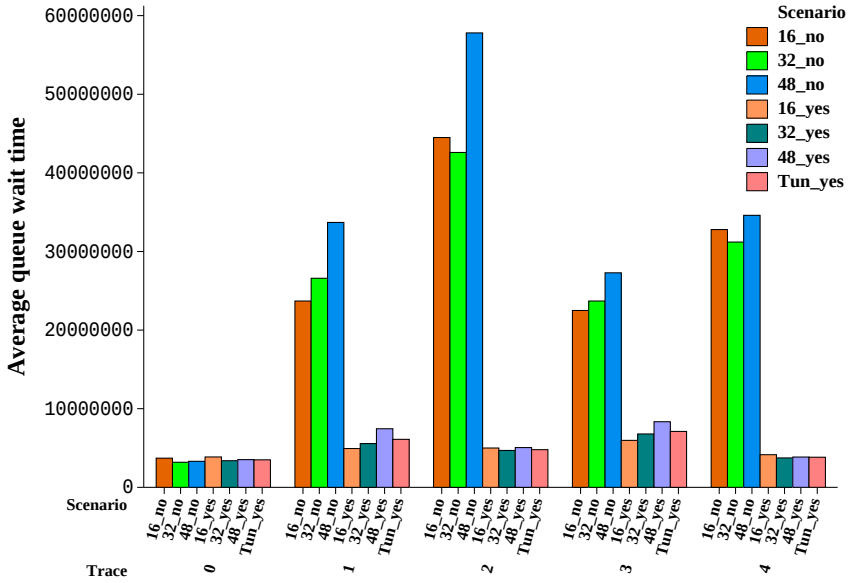


Figure 7.5: Average job queue wait time in various scenarios.

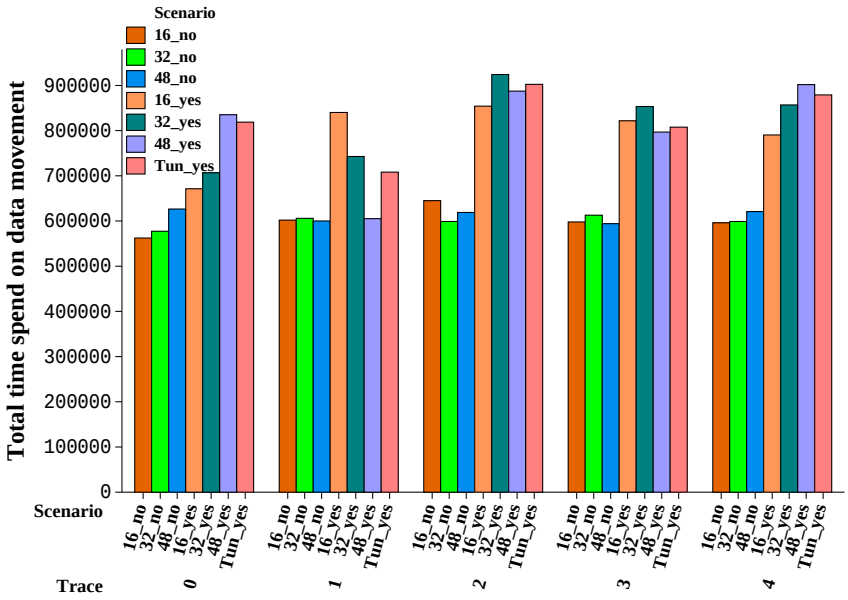


Figure 7.6: Total time spend on data movement in various scenarios.

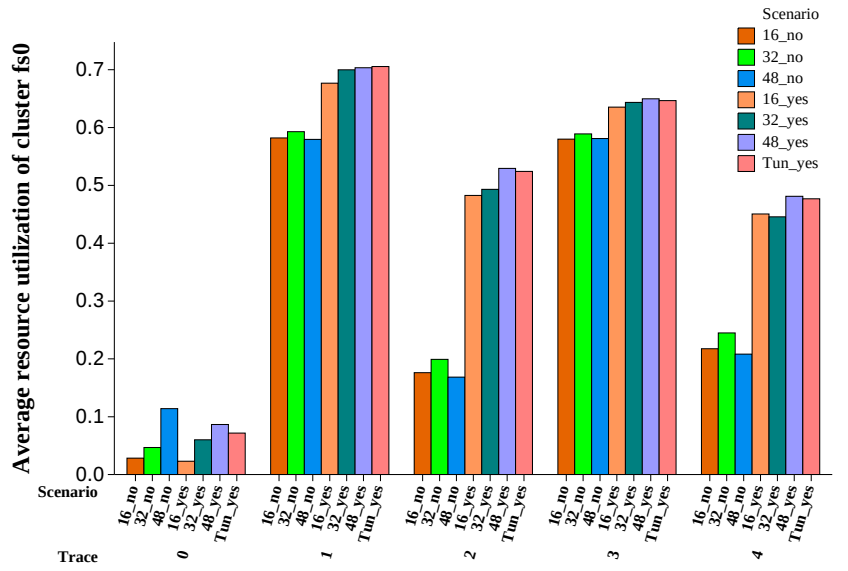
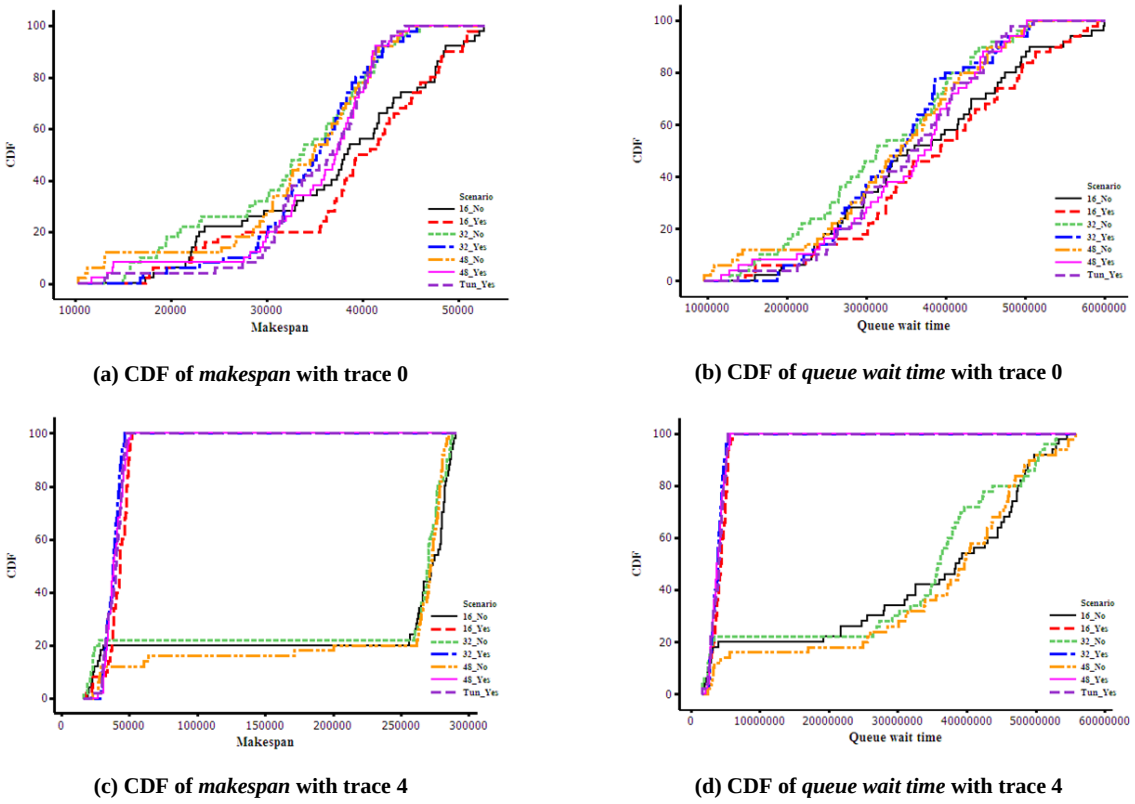


Figure 7.7: Average *resource effective utilization* of cluster fs0 in various scenarios.



(a) CDF of *makespan* with trace 0

(b) CDF of *queue wait time* with trace 0

(c) CDF of *makespan* with trace 4

(d) CDF of *queue wait time* with trace 4

Figure 7.8: CDF of *makespan*, *queue wait time* and *data transfer time* with trace 0 and 4.

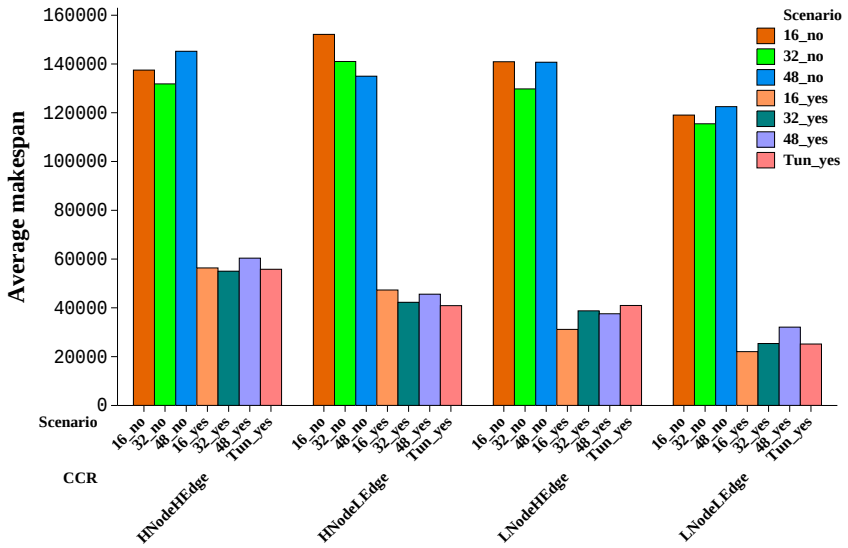


Figure 7.9: Average *makespan* vs. Node- and Edge-weight ratio in various scenarios.

CHAPTER 8
PROTOTYPE

In order to prove the practicality of the proposed scheduling algorithm, a system prototype integrated with Condor/Condor-G [21, 22] is developed. This Chapter first gives a brief introduction about Condor-G architecture in Section 8.1. The design of the prototype design is presented in Section 8.2. Finally, Section 8.2 provides details of the prototype implementation.

8.1 Condor Architecture [21]

Condor provides a rich and varied range of services, which can be simplified into the following three categories:

- Job scheduling: Condor provides means to manage job execution requests as persistent queues of jobs, as well as coordinating and monitoring the remote execution of the jobs on the users behalf. It provides means for users to specify and queue large number of jobs or specify workflow dependencies between jobs.

- Resource management services: A central manager is responsible for collecting resource characteristics and usage information from machines in a Condor pool. It is based on this collected information, and on user priorities, that job requests can be matched to suitable resources for execution.

- Job execution management: Based on matches obtained from the central manager Condor manages the remote execution of jobs on the selected resources. Condor provides the ability to checkpoint jobs

Condor's functionalities have been compartmentalized into a number of individual daemons. Interaction between these daemons is illustrated in Figure 8.1. Particular daemons of interest to us here are the following:

- *condor_schedd*: The Condor scheduler is responsible for maintaining a persistent queue of job execution requests and managing the remote execution of jobs. Jobs are maintained as job *ClassAds* essentially a list of name/expression pairs that represent the various characteristics of a job (input files, arguments, executable, etc.) as well as its requirements and preferences (memory, operating system etc.). The scheduler has been adapted to provide client side job management capabilities for a number of other resource management systems, such as the Globus Toolkit and LSF (Condor-G) [22].

- *condor_collector*: The collector is responsible for maintaining meta-data about all resources and other daemons in a pool in the form of resource *ClassAds*, describing the various characteristics of the resource (memory, current load, Operating system, etc.).

  The *ClassAd* mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (e.g. jobs) with resource offers (e.g. machines). *ClassAds* allows Condor to adopt to nearly any desired resource utilization policy and to adopt a planning approach when incorporating Grid resources. Condor uses matchmaking to bridge the gap between planning and scheduling [110].

8.2 Prototype Design

Condor provides Web Service supports for two primary services: the *Condor scheduler* and the *Condor collector*. The *Condor collector* allows external applications to query a pool collector to determine the type and availability of resources in the pool. It also stores the meta-data about every resource in the pool in the form resource *ClassAds*. This component also allows application retrieve the *ClassAds* through a set of query operations which may

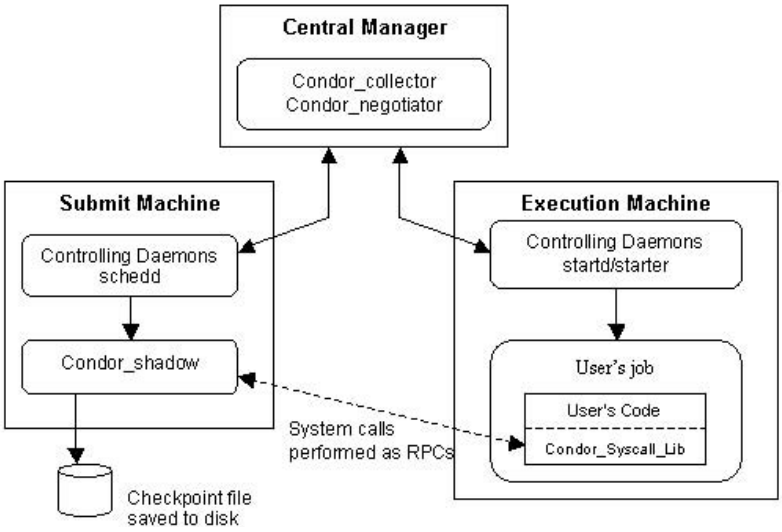


Figure 8.1: Condor architecture overview.

specify particular constraints such as space and operating system type. Particularly, we can map a resource to a job by specifying the resource name as a constraint. The Condor *ClassAd* mechanism will guarantee the job be scheduled to the resource which matches the name defined in the resource constraint.

The *Condor scheduler* Web Service allows applications to submit a job through web service call. The job submission operation creates a new cluster id and job id which can be used later to monitor the job status, such as checking current status of a job. Other queue management operations are provided to cancel, hold or reschedule a job. The *Condor scheduler* also provides means to send input file and binaries to the schedule and retrieve any output files produced through a simple chink-based file transfer protocol.

As illustrated in Figure 8.2, the prototype system design, COllaborative Workflow Scheduler (COWS), consists of the following components:

1. *Submission Manager.* Submission manager can remotely accept any independent jobs or DAGs in predefined XML format. The job submission file describes the input, output and executable files of each job and the dependence among jobs.

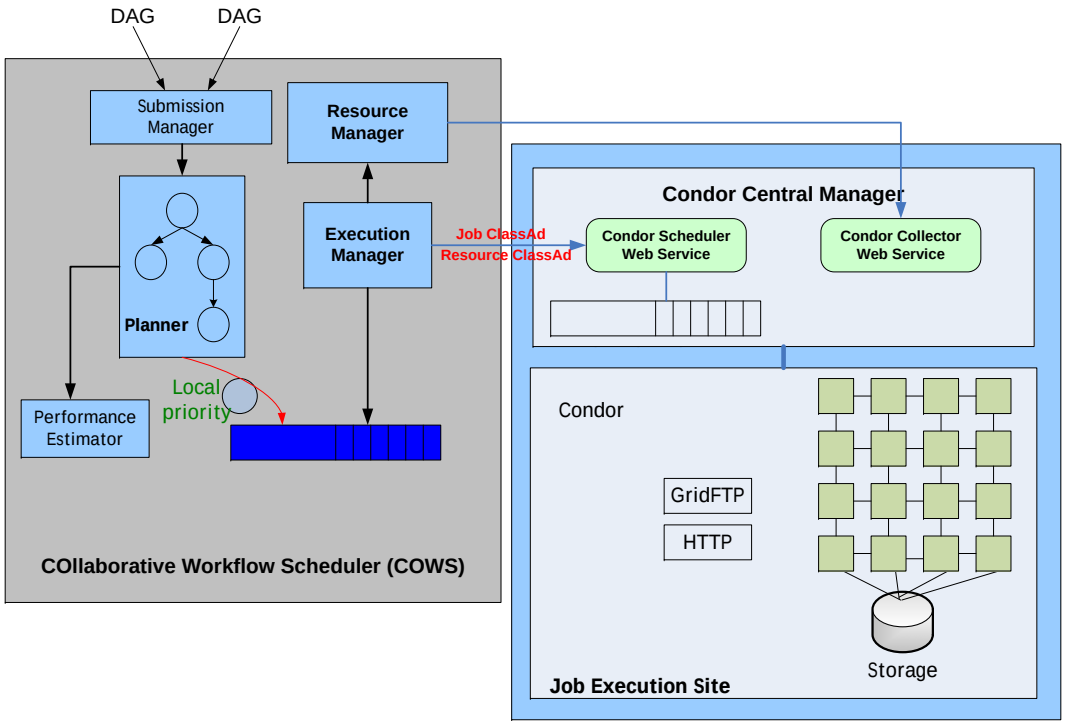


Figure 8.2: COllaborative Workflow Scheduler(COWS) prototype design.

2. *Planner*. The *Planner* ranks each individual jobs and submits the ready to execute jobs to the global pool managed by the *Execution Manager*.

3. *Resource Manager*. This component calls *Condor Collector* Web Service to retrieve the resource information routinely or as demanded by the *Execution Manager*. The *Resource Manager* is responsible for retrieving the information of resources in the Condor pool, including resource attributes and status.

4. *Execution Manager*. This component implements the proposed scheduling algorithm defined in the earlier chapters, maps the resource to each job in the order of global priority. Once the job is mapped to a particular resource, the *Execution Manager* calls the *Condor Scheduler* Web Service to submit the job with predefined resource name in the Job *ClassAd*. A *ClassAd* is is a set of uniquely named expressions used in

Condor. An example of Condor *ClassAd* is illustrated by Figure 8.3. With the resource attribute "Machine" defined in web service call made by the *Execution Manager*, the Condor resource manager will only schedule the job to the named resource. In addition, it also monitors the job status during the workflow execution.

```
MyType          = "Machine"
TargetType      = "Job"
Machine         = "test.cs.wayne.edu"
Arch            = "INTEL"
OpSys           = "SOLARIS251"
Disk            = 35882
Memory          = 512
Requirements    = TARGET.Owner=="MIST"||LoadAvg<=0.5
```

Figure 8.3: An example of Condor *ClassAd*.

8.3 Prototype Implementation

The prototype is implemented in Java 1.5, and leverage three well established frameworks:

- *Event Listen Framework (ELF)* [33]. It supports custom event-listening mechanism in single method listener interface. It is programmatically configurable and customizable to support event-listener for multi-method listener interface. It is used in the prototype to support the collaboration between the *Execution Manager* and the *Planner*. For instance, when a job finishes, the *Execution Manager* will notify the *Planner* of the job status and expect new ready to execute child task being submitted.

- *Apache MINA* [77]. It is a network application framework which helps users develop high performance and high scalability network applications easily. It provides an abstract, event-driven, asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO. This framework is used in the prototype to support remote submission of workflow. In the design, the native port of *condor_schedd* is disabled

with only web service accessible. All jobs, whether it is a workflow or not, will be submitted to the Submission Manager which supports remote submission.

- *Axis* [11]. Axis supports SOAP, which is built on top of HTTP, a foundation layer for web service implementation. In this prototype, we use the Axis to hide the details of the call and generate a wrapper class for the web services, namely, Condor Scheduler and Condor Collector. This is done by taking the WSDL description of the service (condor_schedd.wsdl and condor_collector.wsdl) and generating Java classes that make the low level calls appropriate to building the SOAP requests for each operation, then post-processing the results into the declared return values. Axis also takes note of any URL of the service included in the WSDL and compiles this in to the classes. Thus the client will communicate to the web services like it is a local call.

8.3.1 Package and Class Diagrams

The package structure of the prototype implementation is shown by Figure 8.4. The primary packages are listed below:

- edu.wayne.mist.cows.plan. This package contains all classes implementing the Planner.

- edu.wayne.mist.cows.execute. This package contains all classes supporting the function of the Execution Manager.

- edu.wayne.mist.cows.condor. This package contains the sub classes generated by Axis and a wrapper class which maps to the common operations supported by the web services.

- edu.wayne.mist.cows.main. This package contains the classes implementing job Submission Manager.

- edu.wayne.mist.cows.resource. This package contains the classes supporting Resource Manager.

- edu.wayne.mist.cows.util. This package includes the utility classes which supports data staging and file transfer etc.

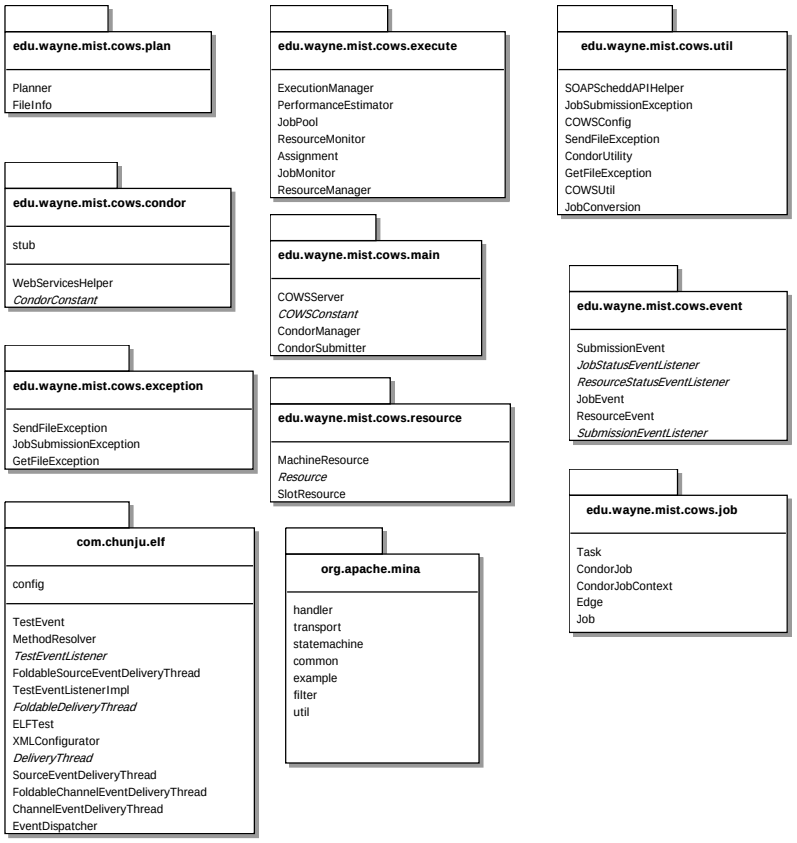


Figure 8.4: Package diagram of COWS.

The class diagram, Figure 8.5, presents the primary Java classes and their relationship. How these classes work together to support workflow scheduling is described in Section 8.3.2 with a sequence diagram.

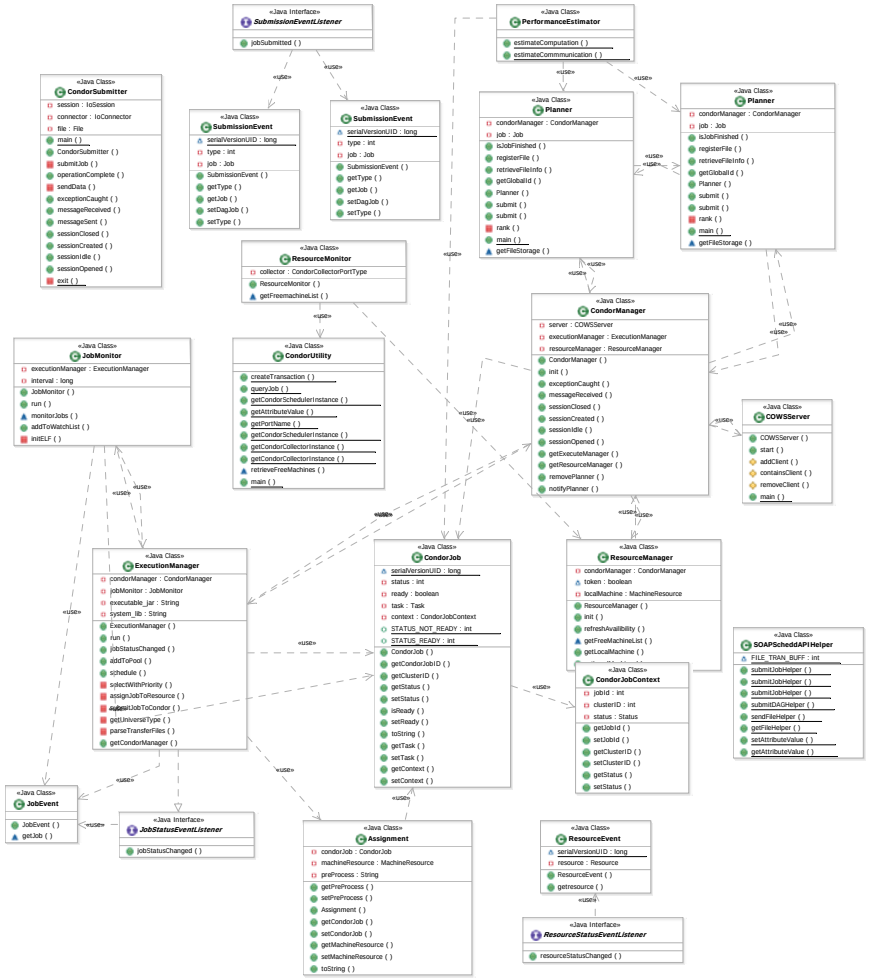


Figure 8.5: Primary classes in COWS.

8.3.2 Sequence Diagram

The sequence diagram of main flow is illustrated by Figure 8.6. When the *CondorSubmitter* receives a DAG from a user, it passes the DAG to the *Planner*. The *Planner* first retrieves the job performance history data and use that to rank all the tasks by applying the algorithm defined in Section 3.2. After ranking the tasks, the *Planner* submits ready to execute tasks to the *ExecutionManager*. Anytime when there are tasks to schedule, the *ExecutionManager* retrieves the free resource list from the *ResourceManager* and assigns the proper resource

to the job in order of priority. The best resource for a job is the one which helps job finish the earliest time. Once a resource is chosen, the *ExecutionManager* will call the *WebServicesHelper* which invokes the corresponding operation provided by Condor Web Services. In order to ask Condor manager to schedule the job to a particular resource which *ExecutionManager* assigns the job to, the web service call explicitly specify the resource attribute "Machine" in the web service call.

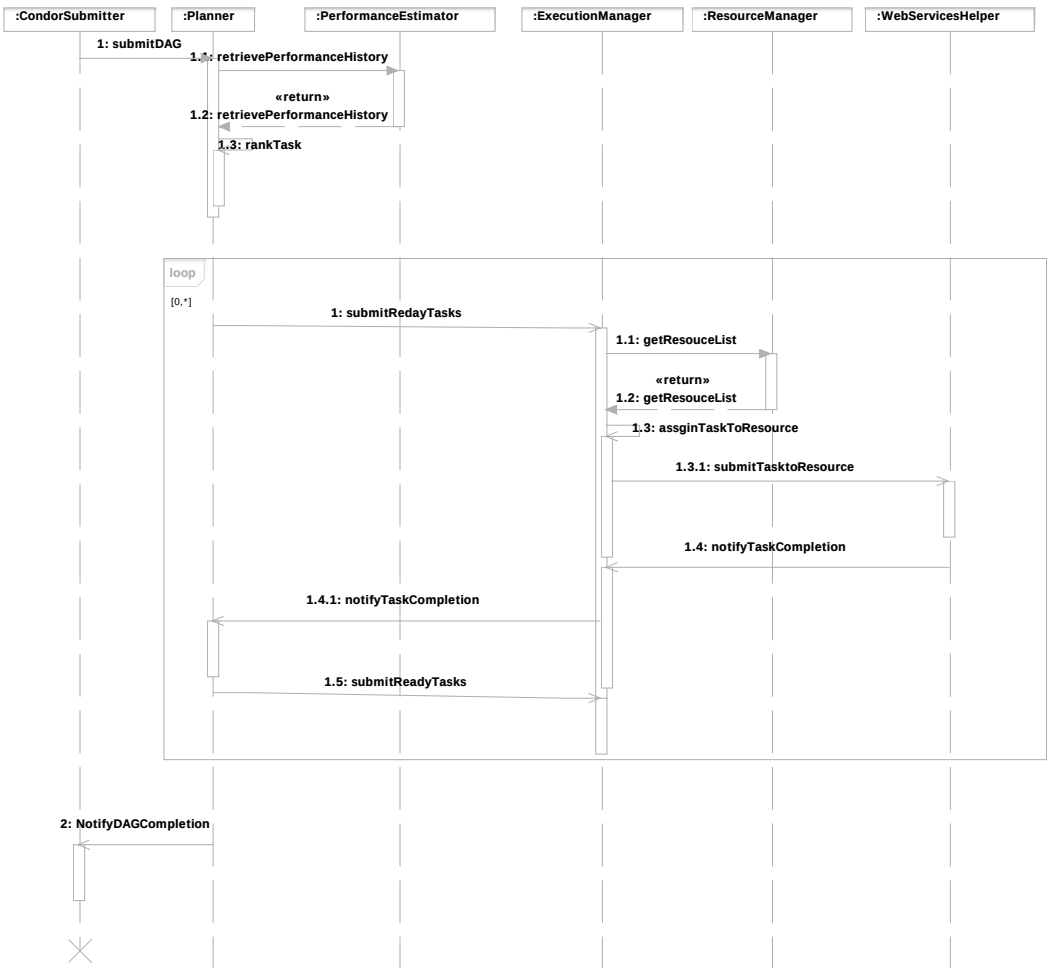


Figure 8.6: Sequence diagram.

8.4 Summary

We present a Java based prototype in this chapter to demonstrate practicability of the scheduling strategy proposed in this dissertation. The prototype shows that the algorithm is implementable on Condor platform via integrating with the Condor Web Services.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

This chapter summarizes the dissertation work in Section 9.1, its limitations in Section 9.2 and finally lays out future work in Section 9.3.

9.1 Summary

This dissertation proposes a hybrid workflow scheduling strategy, namely *planner guided dynamic scheduling*, to address two fundamental challenges in grid workflow scheduling: dynamic workload and dynamic resource.

We first thoroughly review the existing heuristics, their limitations and inability to address these two challenges. As most workflow management systems and the algorithms they employ tend towards two different extremes, either static or dynamic, we believe the proposed hybrid approach, which introduces collaboration between workflow *Planner* and *Executor*, is the answer to the challenges.

As a result, the COllaborative Workflow System(COWS) is designed to implement the proposed strategy and evaluated from multiple perspectives: how to schedule adaptively in an environment where resources dynamically change? how to schedule in a computing environment with dynamic workload? how to schedule in an error prone environment and how to schedule in an extended multi cluster environment? Utilizing published workload traces and workflow test bench, extensive simulation based experiment results prove the soundness of the proposed scheduling strategy.

To further evaluate its practicability, a prototype developed in Java demonstrates that the proposed scheduling strategy can be implemented and integrated with Condor [21].

9.2 Limitations

While extensive evaluation shows that the proposed scheduling strategy outperforms most popular heuristics in simulated based experiments, we believe that it can be further improved if the following limitations are addressed:

- Workflow application performance benchmark. There is no widely accepted performance benchmark for workflow applications, which itself is a great challenge. Researchers use various randomly generated workflows to evaluate their developed heuristics. This dissertation uses the published workflow test bench [54] extensively in order to minimize any possible bias in the evaluation. Even so, a typical real world workflow application always has its own characteristics and uniqueness, for instance the DAG shape and CCR etc.. It is not unusual that a scheduling strategy performs better for one type of workflow than others.

- Failure prediction and queue wait time prediction. This dissertation assumes such prediction services available and ready to use. For example, the QBETS can actually predict the lower bound of job wait time very precisely at confidence level of 95% [91]. But they are still in their infancy stage of practical use. They yet to mature to achieve reasonably high prediction accuracy.

- Assumptions. Even though the scheduling strategy in this dissertation is proposed toward practical implementation, it still makes some assumptions which may not be valid always. For instance, we assume each cluster has a centralized share file storage and the network bandwidth is equal and constant in a local area network without competition.

- Oversimplification of workload and workflow performance modeling. The workload and performance models are oversimplified because they are not focuses of this dissertation. However, as the scheduling strategy is not bound to any specific model,

a model of higher precision can further improve the scheduling effectiveness. In addition, the published workload traces of well known cluster systems are used in the experiments in order to have the evaluation as much objective as possible.

- Multiple objectives. This dissertation does not evaluate the tradeoffs between multi objectives, namely *makespan*, turnaround time, resource utilization and fairness. The evaluation focuses more on the performance metrics, i.e. *makespan* and *turnaround* time, than others. *Resource effective utilization* is measured but only partially from system management perspective. The *fairness* is evaluated qualitatively instead of qualitatively.

9.3 Future Work

The limitations recognized in Section 9.2 rightfully indicate the directions for future work:

- With resource failure prediction and queue wait time prediction technologies being still in progress of maturing, the scheduler should be intelligent enough to leverage predictors of relatively low accuracy. On the other hand, we should continue refining the accuracy requirements in the context of scheduling and provide a practical guidance for further advancement in both failure prediction and queue wait time prediction.

- Recent years have seen considerable progresses made on storage aware, resource availability aware scheduling, which can be utilized to improve the proposed approach in this dissertation. Given that workflow applications typically involve intensive data processing, unnecessary data movement not only harms application performance but also increases extra competition for network and storage resources. For data intensive workflow applications, storage awareness and network connectivity awareness can help improve scheduling effectiveness.

- Workload and performance models should be further refined to better evaluate the effectiveness of a scheduling algorithm, which in turn helps identify the areas for

improvement. However modeling workload and performance itself is a well known great challenge.

- Continue refining the objective function of workflow scheduling to balance the multiple objectives. The performance of a scheduling system should be measurable. Typically, workflow *makespan*, *turnaround* time, *fairness* and *resource effective utilization* are commonly used performance metrics. These objectives are contradictory to each other sometimes and tradeoffs are required. A good objective function can help guide system's behavior and achieve the design goal.

REFERENCES

[1] Dagman. http://www.cs.wisc.edu/condor/dagman/.

[2] Eman. http://blake.bcm.tmc.edu/eman/.

[3] Montage. http://montage.ipac.caltech.edu.

[4] Nsf teragrid. http://www.teragrid.org/.

[5] Open science grid. http://www.opensciencegrid.org/.

[6] Simjava. http://www.dcs.ed.ac.uk/home/hase/simjava/.

[7] Jemal H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society, 2004.

[8] K. Aida and H. Casanova. Scheduling mixed-parallel applications with advance reservations. In *Proceedings of the 17th International Symposium on High-Performance Distributed Computing(HPDC'08)*, pages 65–74, 2008.

[9] J. Almond and D. Snelling. Unicore: uniform access to supercomputing as an element of electronic commerce. *Future Generation Comp. Syst.*, 15(5-6):539–548, 1999.

[10] I. Altintas et al. A framework for the design and reuse of grid workflows. In *Proceeding of 1st International Workshop on Scientific Applications on Grid Computing (SAG'04)*, pages 119–132, 2004.

[11] Axis web service. http://ws.apache.org/axis/.

[12] Mark Bakery and Rajkumar Buyyaz. Cluster computing at a glance. In *High Performance Cluster Computing: Architectures and Systems*, pages 3–47. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[13] F. Berman et al. New grid scheduling and rescheduling methods in the grads project. *International Journal of Parallel Programming*, 33(2):209–229, 2005.

[14] J. Blythe et al. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, Cardiff, UK, 2005.

[15] T. Braun et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel Distribution Computation*, 61(6):810–837, 2001.

[16] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay in space-shared computing environments. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization(IISWC'06)*, pages 213–224, 2006.

[17] L. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73 – 84. Springer US, 2008.

[18] J. Cao, S. Jarvis, S. Saini, and G. Nudd. Gridflow: Workflow management for grid computing. In *Proceeding of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 198–205, 2003.

[19] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 349–363, 2000.

[20] C. Catlett. The philosophy of teragrid: Building an open, extensible, distributed terascale facility. In *Proceeding of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid'02)*, page 8, 2002.

[21] Condor. http://www.cs.wisc.edu/condor.

[22] Condor-g. http://www.cs.wisc.edu/condor/condorg/.

[23] K. Czajkowski, I. Foster, and C. Kesselman. Resource and service management. In *The GRID 2: Blueprint for a new computing infrastructure*, pages 259–283. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[24] H. Dail et al. Scheduling in the grid application development software project. In Jarek Nabrzyski, Jennifer Schopf, and Jan Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[25] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow management in griphyn. In *Grid Resource Management: State of the Art and Future Trends*, pages 99–116. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[26] E. Deelman et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–36, 2003.

[27] E. Deelman et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[28] E. Deelman and Y. Gil. Final report (eds.). In *the NSF Workshop on Challenges of Scientific Workflows, National Science Foundation, Arlington, VA, May 1-2, 2006.*

[29] A. Dogan and Füsun Özgüner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):308–323, 2002.

[30] A. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings of the 11th International Symposium on Parallel Processing(IPPS'97)*, pages 209–218, Washington, DC, USA, 1997. IEEE Computer Society.

[31] A. Downey. Using queue time predictions for processor allocation. In *Proceedings of the Job Scheduling Strategies for Parallel Processing(IPPS'97)*, pages 35–57, London, UK, 1997. Springer-Verlag.

[32] H. El-Rewini and T. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, 1990.

[33] Event listen framework. https://elf.dev.java.net/.

[34] Eurogrid project. http://www.eurogrid.org/.

[35] T. Fahringer et al. Askalon: A grid application development and computing environment. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID'05)*. IEEE, 2005.

[36] T. Fahringer et al. Askalon: a tool set for cluster and grid computing: Research articles. *Concurrent Computing: Practice and Experience*, 17(2-4):143–169, 2005.

[37] I. Foster. What is the grid? a three point checklist. *http://www-fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf*, 2002.

[38] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[39] I. Foster and C. Kesselman. The grid in a nutshell. pages 3–13, 2003.

[40] I. Foster and C. Kesselman. Concept and architecture. In *The GRID 2: Blueprint for a new computing infrastructure*, pages 259–283. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[41] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[42] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 1998. IEEE Computer Society.

[43] J. Frey et al. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[44] J. Frey and T. Tannenbaum. Ogf 19 - condor software forum - condor-g. In *The 19th Open Grid Forum - OGF19*, 2007.

[45] S. Fu and C. Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'07)*, 2007.

[46] M. Garey and D. Johnson. *Computers and Intractibility: A guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[47] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, 2007.

[48] The globus alliance. http://www.globus.org/.

[49] Grid application development software (grads) project. http://www.hipersoft.rice.edu/grads/.

[50] Griphyn. http://www.griphyn.org/.

[51] L. He, S. Jarvis, D. Spooner, and G. Nudd. Performance evaluation of scheduling applications with dag topologies on multiclusters with independent local schedulers. In *in Proc. of 20th IEEE International Symposium of Parallel and Distributed Processing Symposium(IPDPS'06)*, pages 8 – 15, 2006.

[52] L. He, S. Jarvis, D. Spooner, and G. Nudd. Performance evaluation of scheduling applications with dag topologies on multiclusters with independent local schedulers. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium(IPDPS'06)*, Rhodes Island, Greece, 2006.

[53] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[54] U. Hönig and W. Schiffmann. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS'04)*. IEEE, 2004.

[55] U. Hönig and W. Schiffmann. A meta-algorithm for scheduling multiple dags in homogeneous system environments. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*. IEEE, 2006.

[56] R. Huang, H. Casanova, and A. Chien. Automatic resource specification generation for resource selection. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[57] S. Hunold, T. Rauber, and G. Runger. Dynamic scheduling of multi-processor tasks on clusters of clusters. In *the Proceedings of 2007 IEEE International Conference on Cluster Computing*, pages 507–514, Washington, DC, USA, 2007. IEEE Computer Society.

[58] S. Hunold, T. Rauber, and F. Suter. Scheduling dynamic workflows onto clusters of clusters using postponing. In *the Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 669–674, Washington, DC, USA, 2008. IEEE Computer Society.

[59] J. Hwang et al. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[60] S. Hwang and C. Kesselman. Gridworkflow: A flexible failure handling framework for the grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

[61] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521, 1996.

[62] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[63] G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):645–662, 2001.

[64] Lead. https://portal.leadproject.org/.

[65] H. Li, D. Groep, and L. Wolters. Workload characteristics of a multi-cluster supercomputer. In *Proceeding of 10th International WorkshopJob Scheduling Strategies for Parallel Processing (JSSPP'04)*, pages 176–193, 2004.

[66] Y. Li et al. Fault-driven re-scheduling for improving system-level fault resilience. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP'07)*, page 39, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Y. Li and Z. Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 531–538, Washington, DC, USA, 2006. IEEE Computer Society.

[68] Yinglung Liang, Anand Sivasubramaniam, and Jose Moreira. Filtering failure logs for a bluegene/l prototype. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 476–485, Washington, DC, USA, 2005. IEEE Computer Society.

[69] J. Liou and M. Palis. Cass: an efficient task management system for distributed memory architectures. In *The Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, page 289, Washington, DC, USA, 1997. IEEE Computer Society.

[70] M. Lopez, E. Heymann, and M. Senar. Analysis of dynamic heuristics for workflow scheduling on grid systems. In *Proceedings of the 5th International Symposium on Parallel and Distributed Computing (ISPDC'06)*, pages 199–207. IEEE, 2006.

[71] Los alamos national laboratory. operational data to support and enable computer science research, 2006.

[72] Platform lsf. http://www.platform.com/Products/platform-lsf.

[73] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 59(2):107–131, 1999.

[74] G. Malewicz, I. Foster, A. Rosenberg, and M. Wilde. A tool for prioritizing dagman jobs and its evaluation. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 156–167. IEEE, 2006.

[75] A. Mandal et al. Scheduling strategies for mapping application workflows onto the grid. In *Proceeding of the 14th International Symposium on High Performance Distributed Computing (HPDC'05)*, pages 125–134. IEEE, 2005.

[76] A. McGough et al. Making the grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.

[77] Apache mina network application framework. http://mina.apache.org/.

[78] Moab cluster suite. http://www.clusterresources.com/.

[79] T. N'Takpe, F. Suter, and H. Casanova. A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 35, Washington, DC, USA, 2007. IEEE Computer Society.

[80] D. Nurmi, J. Brevik, and R. Wolski. Qbets: Queue bounds estimation from time series. In *Proceedings of the 13th International Workshop of Job Scheduling Strategies for Parallel Processing(JSSPP'07)*, pages 76–101, 2007.

[81] D. Nurmi, J. Brevik, and R. Wolski. Qbets: queue bounds estimation from time series. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems(SIGMETRICS'07)*, pages 379–380, 2007.

[82] D. Nurmi, A. Mandal, J. Brevik, C. Koelbel, R. Wolski, and K. Kennedy. Evaluation of a workflow scheduler using integrated performance modelling and batch queue wait time prediction. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing(SC'06)*, page 119, New York, NY, USA, 2006. ACM.

[83] T. Oinn et al. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinfomatics*, 20(17):3045–3054, 2004.

[84] A. Oliner et al. Fault-aware job scheduling for bluegene/l systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium(IPDPS'04)*. IEEE Computer Society, 2004.

[85] A. Oliner, R. Sahoo, J. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 299.2, Washington, DC, USA, 2005. IEEE Computer Society.

[86] Portable batch system. http://www.openpbs.org.

[87] D. Oppenheimer et al. Service placement in shared wide-area platforms. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP'05)*, pages 1–1, New York, NY, USA, 2005. ACM.

[88] Pbs:portable batching system. http://www.pbsgridworks.com/.

[89] Pegasus (planning for execution in grids). http://pegasus.isi.edu/.

[90] Planet lab. http://www.planet-lab.org.

[91] Qbets web service. http://spinner.cs.ucsb.edu/batchq/.

[92] A. Radulescu and A.Gemund. Fast and effective task scheduling in heterogeneous systems. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, page 229, Washington, DC, USA, 2000. IEEE.

[93] T. Rauber and G. R. M-task-programming for heterogeneous systems and grid environments. *Parallel and Distributed Processing Symposium, International*, 5:178b, 2005.

[94] X. Ren et al. Prediction of resource availability in fine-grained cycle sharing systems empirical evaluation. *Journal of Grid Computing,*.

[95] M. Romberg. The unicore architecture: Seamless access to distributed resources. In *Proceeding of the 8th International Symposium on High Performance Distributed Computing*, pages 287–293, 1999.

[96] M. Romberg. The unicore grid infrastructure. *Sci. Program.*, 10(2):149–157, 2002.

[97] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environement. In *9th International Workshop of Job Scheduling Strategies for Parallel Processing (JSSPP'03)*, pages 87–104, 2003.

[98] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004.

[99] F. Salfner and M. Malek. Proactive fault handling for system availability enhancement. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing*

Symposium (IPDPS'05) - Workshop 16, page 281.1, Washington, DC, USA, 2005. IEEE Computer Society.

[100] F. Salfner, M. Schieschke, and M. Malek. Predicting failures of computer systems: a case study for a telecommunication system. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[101] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[102] B. Schroeder and G. Gibson. Understanding failures in petascale computers. *Journal of Physics: Condensed Matter*, 19(45), 2007.

[103] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transaction of Parallel Distribtion System*, 4(2):175–187, 1993.

[104] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proceedings of the IEEE Heterogeneous Computing Workshop (HCW'96)*, 1996.

[105] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proceedings of the Job Scheduling Strategies for Parallel Processing(JSSPP'99)*, pages 202–219, London, UK, 1999. Springer-Verlag.

[106] D. Sulakhe et al. Gnare: an environment for grid-based high-throughput genome analysis. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 455–462, 2005.

[107] A. Sulistio, U. Cibej, S. Venugopal, B. Robic, and R. Buyya. A toolkit for modelling and simulating data grids: an extension to gridsim. *Concurr. Comput. : Pract. Exper.*, 20(13):1591–1609, 2008.

[108] A. Sulistio, W. Schiffmann, and R. Buyya. Advanced reservation-based scheduling of task graphs on clusters. In *Proceedings of the 13th Annual IEEE International Conference on High Performance Computing (HiPC'06)*, Berline, Germany, 2006. Springer Verlag.

[109] F. Suter, F. Desprez, and H. Casanova. From heterogeneous task scheduling to heterogeneous mixed parallel scheduling. In *the Proceedings of 10th International Euro-Par Parallel Processing Conference*, pages 230–237, 2004.

[110] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc, 2002.

[111] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distribution Systems*, 13(3):260–274, 2002.

[112] M. Wieczorek, A. Hoheisel, and R. Prodan. Taxonomies of the Multi-criteria Grid Workflow Scheduling Problem. In *Proceedings of the CoreGRID Workshop on Grid Middleware*, Dresden, Germany, June 2007. Springer-Verlag.

[113] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Record*, 34(3):56–62, 2005.

[114] Wien2k. http://www.wien2k.at/.

[115] R. Wolski, N. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comput. Syst.*, 15(5-6):757–768, 1999.

[116] M. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, 1990.

[117] M. Wu, W. Shu, and H. Zhang. Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 375–385, 2000.

[118] P. Yalagandula et al. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *Proceedings of the Workshop on Real, Large Distributed Systems (WORLDS'04)*, 2004.

[119] L. Yang, J. Schopf, and I. Foster. Anomaly detection and diagnosis in grid environments. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2007.

[120] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.

[121] H. Yu et al. Plan switching: an approach to plan execution in changing environments. In *Proceeding of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, page 14, April 2006.

[122] Z. Yu and W. Shi. An adaptive rescheduling strategy for grid workflow applications,. In *Proceeding of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, Florida, USA, March 2007.

[123] Z. Yu and W. Shi. A planner-guided scheduling strategy for multiple workflow applications. *Proceedings of the Fourth International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems, in conjunction with ICPP 2008*, pages 1–8, 2008.

[124] Y. Zhang et al. Performance implications of failures in large-scale cluster scheduling. In *Proceedings of 10th International WorkshopJob Scheduling Strategies for Parallel Processing (JSSPP'04)*, pages 233–252, 2004.

[125] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings of the 15th Heterogeneous Computing Workshop (HCW)*, Rhodes Island, Greece, April 2006.

ABSTRACT

TOWARD PRACTICAL MULTI-WORKFLOW SCHEDULING IN CLUSTER AND GRID ENVIRONMENTS

by

ZHIFENG YU

May 2009

Advisor: Dr. Weisong Shi

Major: Computer Science

Degree: Doctor of Philosophy

Workflow applications are gaining popularity in recent years because of the prevalence of cluster and Grid environments. Many algorithms have been developed ever since, however two fundamental challenges in this area, i.e., dynamic resource and dynamic workload, are not well addressed. In cluster and Grid environments, resources may be contributed and controlled by different virtual organizations and shared by a variety of users who in turn submit various kinds of applications. Resources are heterogeneous under different ownership, their availability varies over time and may fail in a high rate. On the other hand, resources are shared and hence competed among many applications with various computation requirements. Existing static algorithms are designed to schedule a single workflow application, without considering other workloads and any resource competition in the system. Hence static approaches are not utilized widely in practice despite its known advantages. Dynamic scheduling approaches can handle the dynamic workload and resources practically by nature but their effectiveness has yet to optimize as they do not have a global view of workflow application and scheduling decision is made nearsighted locally.

In this dissertation, as an effort toward practically scheduling workflow applications in cluster and Grid environments, a failure aware dynamic scheduling strategy for multiple workflow applications is proposed. The approach makes scheduling decision only when a task

is ready, as traditional dynamic approach does, but leverages task dependency information, execution time estimation, failure prediction and queue wait time prediction. With pre-assigned priority for each task by the workflow *Planner*, the workflow *Executor* globally prioritizes all the ready to execute tasks in queue and schedules the individual task to the most suitable resource collection in order to minimize the overall workflow execution time. Furthermore, the algorithm is extended to a cluster of clusters environment, where each cluster has its own local workload management system. As a conclusion, the findings of the research is four folded:

1. With adaptability to dynamic resource change, the proposed strategy not only outperforms the purely dynamic ones but also improves over the traditional static ones. And it performs more efficiently with data intensive application of higher degree of parallelism.

2. When guided by the *Planner*, the proposed strategy can schedule multiple workflows dynamically without requiring merging the workflows*a priori*. It significantly outperforms two other traditional dynamic algorithms by 43.6% and 36.7% with respect to workflow *makespan* and *turnaround* time respectively, and it performs even better when the number of concurrent workflow applications increases and the resources are scarce.

3. We observer that the traditional failure prediction accuracy definitions impose different performance implications on different applications and fail to measure how that improves scheduling effectiveness, and propose two definitions on failure prediction accuracy from the perspectives of system and scheduling respectively. The comprehensive evaluation results using real failure traces show that the proposed strategy performs well with practically achievable prediction accuracy by reducing the average *makespan*, the *loss time* and the *number of job rescheduling*.

4. The proposed algorithm can be augmented to Grids in form multicluster where each cluster has its own workload management system. The proposed queue wait time aware algorithm leverages the advancement of queue wait time prediction techniques and empirically studies if the tunability of resource requirements helps scheduling. The extensive experiment with both real workload traces and test bench shows that the queue wait time aware algorithm improves workflow performance by 3 to 10 times in terms of average *makespan* with relatively very low cost of data movement.

Finally, the research studies how to benefit from existing researches and practices on both static and dynamic scheduling, introduces a hybrid scheduling scheme, i.e., a planner guided dynamic scheduling approach, targets on dynamic workload on cluster and Grid environment. A prototype is developed based on Condor platform to prove the concept of proposed algorithm.

AUTOBIOGRAPHICAL STATEMENT

Zhifeng Yu is a Ph.D. student of computer science at Wayne State University. His current research focuses on workflow scheduling in cluster and grid environment, software engineering, service engineering and project management. He received his Bachelor degree in Mathematics and Masters degree in Engineering Project Management from Tongji University of China in 1990 and 1993 respectively, and Masters degree in Computer Science from Wayne State University in 1998.

Selected publications

1. **Zhifeng Yu** and Weisong Shi, Queue Wait Prediction Enabled Dynamic Scheduling of Workflows on Multi Clusters, submitted to CCGrid'09

2. Jayashree Ravi, **Zhifeng Yu** and Weisong Shi, A Survey on Dynamic Web Content Generation and Distribution Techniques, submitted to Journal of Network and Computer Applications

3. **Zhifeng Yu**, Chenjia Wang and Weisong Shi, FLAW: Failure-Aware Workflow Scheduling in High Performance Computing Environments, submitted to Journal of Cluster Computing

4. **Zhifeng Yu** and Weisong Shi, A Planner-Guided Scheduling Strategy for Multiple Workflow Applications, in the Proceedings of the fourth International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems, in conjunction with ICPP 2008, 2008

5. **Zhifeng Yu** and Weisong Shi, An Adaptive Rescheduling Strategy for Grid Workflow Applications, in Proceedings of the 21st International Parallel and Distributed Processing Symposium, 2007

6. **Zhifeng Yu** and Vaclav Rajlich, Hidden Dependencies in Program Comprehension and Change Propagation, in the Proceedings of IEEE International Workshop on Program Comprehension, 2001